



Karlsruhe Institute of Technology

Institut für Angewandte Informatik und
Formale Beschreibungsverfahren (AIFB)



SAP AG

SAP Research, CEC Karlsruhe

Semantic Service Discovery using Natural Language Queries

Diploma Thesis

Prof. Dr. Rudi Studer

Institute of Applied Informatics
and Formal Description Methods AIFB
Karlsruhe Institute of Technology

of

cand. inform.

Marc Mültin

Supervisors:

Prof. Dr. Rudi Studer (AIFB)

Veli Bicer (FZI)

Paul Peitz (SAP Research CEC Karlsruhe)

Started on: June 01, 2008

Submitted on: February 15, 2009

Address:

Marc Mültin

Klosterweg 28/G402

76131 Karlsruhe

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, _____

Zusammenfassung, deutsch

Das Internet, wie wir es heute kennen, kann als die größte globale Datenbank angesehen werden. Zu Beginn bestand es nur aus einzelnen Webseiten, welche über sogenannte Hyperlinks miteinander verbunden waren. Diese Form des Internets wird auch als *Web of pages* bezeichnet. Aber es hat sich weiterentwickelt zu einem Internet, welches auch aktivere Komponenten beinhaltet, die Rede ist hier von *Web Services*.

In seinem Artikel im Magazin Scientific American aus dem Jahre 2001 stellt Tim Berners-Lee, der Erfinder des Internets, fest, dass es dem heutigen Internet an einer semantischen Struktur fehlt, welche den Zusammenhang seiner Komponenten festhält. Basierend auf diesem Artikel ist die Vision des Semantic Web eine, in der durch Ontologien angereicherte semantische Beschreibungen überall verfügbar sein werden. Hierdurch soll anspruchsvolle Interoperabilität zwischen Agenten beispielsweise im E-Commerce-Bereich ermöglicht werden. Die semantische Beschreibung von Web Services wurde unter anderem dadurch angeregt, die serviceorientierte elektronische Datenverarbeitung zu einem nächst höheren Level zu führen, wodurch zeitaufwändige Programmiertätigkeiten semi-automatisiert werden sollen.

Das übergeordnete Ziel dieser Arbeit ist die Ermöglichung einer Suche nach Web Services, die von SAP in ihrem Enterprise Service Repository bereit gestellt werden, mit Hilfe von semantischen Technologien. Wir entscheiden uns für ein semantisches Suchwerkzeug welches uns ermöglicht, Anfragen in natürlicher Sprache zu stellen. Somit soll verhindert werden, dass der normale Benutzer weder ein fundiertes Wissen über SQL-ähnliche Anfragesprachen noch über den Inhalt der benutzten semantischen Informationen (Ontologien) mitbringen muss. Denn dies würde ansonsten die Akzeptanz der Software erheblich verringern. Wir testen unser Suchwerkzeug mit diversen Ontologien, um herauszufinden, wie geeignet unsere Software bezüglich Reasoning und Portabilität ist.

Abstract

The current Web as we know it today can be assumed to be the biggest global database. Beginning with just pages on the Web being related to each other via hyperlinks (also referred to as the *Web of pages*), the Internet has further developed and enhanced to also include *more active components*, namely so-called *Web Services*. In his article on Scientific American in 2001, Tim Berners-Lee states that the Web today lacks the existence of a semantic structure to keep the interdependency of its components. According to this article, the Semantic Web vision is one in which rich, *ontology-based semantic markup* becomes widely available, enabling sophisticated interoperability among agents, e.g., in the e-commerce area. The semantic description of Web Services was proposed amongst others to take service oriented computing to a new level by allowing to semi-automate time-consuming programming tasks.

The overall *goal* of this work is to facilitate the search for Web Services provided by SAP in its Enterprise Service Repository with the help of semantic technologies. We choose to use a semantic search tool which allows for queries formulated in natural language, thus avoiding the need of extensive knowledge about the semantic data being used or SQL-like query languages such as SPARQL, as this would lower the acceptance for a naive user. We test our question answering tool against several ontologies to find out how capable this tool is regarding reasoning power, expressivity and portability.

Acknowledgements

I want to thank my colleagues at SAP for their great support and help throughout my work.

Big thanks also to Vanessa Lopez, the developer of AquaLog, who spent a lot of time explaining AquaLog to me and fixing bugs I discovered during my research.

I also want to thank my advisors, Veli Bicer (FZI) and Paul Peitz (SAP), for their advises and support.

I for sure want to thank my friends who supported me during harsh times and also proofread my work.

Last but not least, I definitely want to thank my parents who supported me throughout my studies at the University of Karlsruhe (TH) and made this diploma thesis possible.

Contents

Abstract	vii
Acknowledgements	ix
List of Figures	xiii
List of Abbreviations	xv
List of Abbreviations	xv
1 Introduction	1
1.1 Motivation, Goals and Approach	1
1.2 Requirements	3
1.3 Thesis Structure	5
2 Current Web Technologies	7
2.1 Web Services	8
2.2 The Web Service Technology Stack	9
2.3 WSDL	11
2.4 UDDI	13
3 Step into the Semantic World	17
3.1 The Vision of the Semantic Web	17
3.2 Semantic Web Services	19
3.3 Information Retrieval and Service Discovery	19
3.4 Mechanisms of Service Discovery	21
3.5 Ontologies	22
3.6 Knowledge Bases vs. Ontologies	24
3.7 RDF	24
3.7.1 RDFS - RDF Schema	26
3.7.2 Sesame	27
3.8 OWL	27
3.9 Ontology-based Discovery Approaches	28
3.10 Reasoning	30
3.10.1 Subsumption Reasoning	31
3.11 Quality of Ontology Design	32

4	Question Answering	35
4.1	AquaLog	36
4.1.1	An Illustrative Example	37
4.1.2	AquaLog's Helpers	40
4.1.2.1	GATE	40
4.1.2.2	WordNet	41
4.2	SeRQL	41
5	Design and Adaptation	43
5.1	The Global Architecture	43
5.1.1	From Questions to Query Triples	45
5.1.2	From Triples to Answers	48
5.2	Query Answering Limitations	51
5.3	The Triple Approach Problem	52
5.3.1	Blank Nodes	55
5.3.2	Enhancing reasoning capabilities	57
5.3.2.1	TRREE Engine	58
5.3.2.2	The SAIL API	59
5.4	Custom Rule-Sets	60
5.4.1	Semantics Supported by Default	61
6	Implementation	63
6.1	SwiftOWLIM SAIL Configuration	63
6.2	Performance Optimisation Parameters	65
6.3	Custom Rule-Set Creation	66
6.4	Problems with SeRQL-Directives	72
6.5	Applicability of the TEXO Ontology	75
6.6	Customising Configuration Files	78
6.7	Applicability of the Parameter Ontology	80
7	Summary and Conclusions	81
8	Outlook	83
A	Appendix A	85
B	Appendix B	87
C	Appendix C	91
	Bibliography	93
	Index	96

List of Figures

2.1	Service	8
2.2	The service lifecycle	9
2.3	The Web Service Technology Stack	10
2.4	A representation of concepts defined by a WSDL 1.1 document	12
2.5	The Web Services Framework	14
3.1	The Semantic Web technology stack	18
3.2	The vision of the Semantic Web	20
3.3	Recall and Precision	20
3.4	A graphical representation of a business trip ontology	23
3.5	A simple RDF graph to describe the relationship between a book and the publishing house Springer	25
3.6	An example RDF graph	26
4.1	The AquaLog data model	38
4.2	Illustrative example of user interactivity in AquaLog	39
4.3	Illustrative example of AquaLog disambiguation	40
5.1	The overall architecture of our approach	44
5.2	Example of GATE annotations and linguistic triples for basic queries	47
5.3	Example of GATE annotations and linguistic triples for basic queries with clauses	48
5.4	Example of AquaLog in action for basic generic-type queries	50
5.5	Example of RSS in action for relations formed by a concept	51
5.6	The structure of the pizza ontology as illustrated by Protégé	53
5.7	Margherita pizza modelled in OWL	54
5.8	AquaLog is unable to satisfactorily answer a simple question for pizza toppings	54

5.9	AquaLog aks for user feedback to disambiguate a relation term	55
6.1	Configuration of the SAIL stack for the SwiftOWLIM SAIL	64
6.2	SeRQL query results before application of OWLIM rule	69
6.3	SeRQL query results after application of OWLIM rule	70
6.4	Hierarchical view of the pizza ontology in AquaLog	72
6.5	The TEXO ontology as illustrated by Protégé	76
6.6	Information about the eco-calculator presented by AquaLog	76
6.7	Answer drawn from the TEXO ontology	77
6.8	Answer drawn from the TEXO ontology	77
6.9	Information about the instance <code>cde_provider</code>	78

List of Abbreviations

API	A pplication P rogramming I nterface
DBMS	D ata B ase M anagement S ystem
DL	D escription L ogics
GATE	G eneral A rchitecture for T ext E ngineering
KB	K nowledge B ase
KR	K nowledge R epresentation
NL	N atural L anguage
NLP	N atural L anguage P rocessing
OWL	W eb O ntology L anguage
QA	Q uestion A nswering
QoS	Q uality of S ervice
RDF	R esource D escription F ramework
RQL	R DF Q uery L anguage
RSS	R elation S imilarity S ervice
SAIL	S torage A nd I nterface L ayer
SeRQL	S esame R DF Q uery L anguage
SOA	S ervice- O riented A rchitecture
SOAP	S imple O bject A ccess P rotocol
TRREE	T riple R easoning and R ule E ntailment E ngine
UDDI	U niversal D escription D iscovery and I ntegration
URI	U niform R esource I dentifier
URL	U niform R esource L ocator
W3C	W orld W ide W eb C onsortium
WSD	W eb S ervice D escription
WSDL	W eb S ervice D escription L anguage
XML	E xtensible M arkup L anguage
XSD	X ML- S chema- D efinition

"Getting information off the Internet is like taking a drink from a fire hydrant."

- Mitchell Kapor

1

Introduction

This diploma thesis aims at facilitating an easy, intuitive way to query for Web Services - which are described by using semantic technologies - and introduces all the software and methodologies needed to achieve this overall goal. The introductory chapter explains motivation, goals and our very own approach to the solution of the given problem. An outline of the remaining chapters is given at the end of chapter one.

1.1 Motivation, Goals and Approach

The current Web as we know it today can be assumed to be the biggest global database. Starting off in 1989, when its inventor Tim Berners-Lee launched the World Wide Web project while working at the European Organisation for Nuclear Research (CERN), it was primarily an academic, technical Internet. It took about half a decade for the Internet to gain public interest and thus laying the groundwork for its evolving nature ever since. Not only the network itself evolved in its size, but the content published over the Internet as well. Beginning with just pages on the Web¹ being related to each other via hyperlinks (also referred to as the *Web of pages*), the Internet has further developed and enhanced to also include *more active components*, namely so-called *Web Services*².

John Naisbitt remarks in [Nais88], "We are drowning in data but starving for information", which has become a dictum nowadays as more and more data has spread over the internet, but it is getting harder and harder to exactly find what we are

¹Note that we use the terms *Web* and *Internet* synonymously.

²Web Services are elaborated on in section 2.1

looking for and get precise answers.

Berners-Lee, who has served as the W3C³ Director since W3C was founded in 1994, has a vision of the *Semantic Web* which he first outlined in his article on Scientific American in 2001 [BLHL01]. There he states that the Web today lacks the existence of a semantic structure to keep the interdependency of its components: "The Semantic Web will bring structure to the meaningful content of Web pages, creating an environment where software agents roaming from page to page can readily carry out sophisticated tasks for users."

According to this article, the Semantic Web vision is one in which rich, *ontology-based*⁴ *semantic markup* becomes widely available, enabling sophisticated interoperability among agents, e.g., in the e-commerce area, and to support human web users in locating and making sense of information. Thus, the availability of semantic markup on the web opens the way to novel, sophisticated forms of *question answering*, which can potentially provide increased precision and recall⁵ compared to today's search engines.

The semantic description of Web Services (SWS) was proposed amongst others to take service oriented computing to a new level by allowing to semi-automate time-consuming programming tasks, as Küster et al. [Grad08] state: "At the core of SWS are solutions to the problem of SWS matchmaking, e.g., the problem of comparing a set of semantic service advertisements with a semantic request description to determine those services that are able to fulfil the given request."

The overall *goal* of this work is to facilitate the search for Web Services provided by SAP in its Enterprise Service Repository with the help of semantic technologies.

The descriptions of these Web Services have been analysed in a prior work [Zalt08] to generate a service ontology for the SAP domain. With the help of resources such as the SAP-own glossary SAPterm [AG] and Business Object⁶ descriptions, a *parameter ontology* was created which serves as a starting point for our work.

A variety of semantic technology standards, namely i.e. RDF(S) [BrGu04], OWL [W3C04], as well as query languages for semantic data such as SPARQL [W3Cb] have been developed in order to create and work with semantically enhanced data. However, a user who is not familiar with these technologies would face high barriers if he or she would have to learn e.g. the syntax of SPARQL prior to the formulation of a semantic request. In order to introduce a query front-end to a broad variety of users, there must be an intuitive access to the semantic content for users, thus the query front-end should behave as an interface between *natural language queries* and semantic request descriptions. This is one of our major goals pursued throughout this work.

³<http://www.w3.org/>

⁴The term *ontology* will be explained in section 3.5

⁵The terms *precision* and *recall* are further explained in section 3.3

⁶Business Objects represent real-world entities which are - with relation to business - significant for a company

To accomplish this requirement, we came across a remarkable software system during our research, called AquaLog⁷. It is a portable open-source question-answering system which takes queries expressed in natural language and an ontology as input, and returns answers drawn from this ontology. It is portable in the sense that the AquaLog system allows the user to choose an ontology and then ask queries with respect to the universe of discourse covered by the ontology (essentially a semantic viewpoint)⁸. It makes use of the GATE⁹ NLP platform, string metric algorithms, WordNet¹⁰ and a novel ontology-based *Relation Similarity Service* to make sense of user queries with respect to the target ontology.

Thus, AquaLog is especially suitable as a front-end software to organisational semantic intranets where an organisational ontology is used as the basics for semantic markup. This being said, we will show in this work how this tool will help us to achieve the goals of this diploma thesis.

To further enhance the reasoning capabilities of AquaLog, another tool called SwiftOWLIM¹¹ is being used, in order to make use of reasoning not only for RDF(S) files but also for ontologies modelled in the OWL language (as is the parameter ontology mentioned above). To be more precise, SwiftOWLIM is a plug-in for Sesame¹² which is one of the most popular semantic repositories that supports RDF(S) and all the major syntaxes and query languages related to it, according to [BrKH02]. AquaLog closely collaborates with the Sesame framework, and hence, AquaLog, SwiftOWLIM and Sesame will be the major tools enabling us to reach our goals.

Since the parameter ontology delivered by the prior work needs to be adjusted to properly formulate queries and get meaningful results in return, we will also take a look at how this adjustment could be achieved.

Finally, we will evaluate our approach and have a look at further enhancement and optimisation possibilities.

1.2 Requirements

The scope of our work is to facilitate the discovery of Web Services provided by SAP by making use of semantic technologies. We want to achieve that a user will be able to search for these Web Services in an intuitive way.

These Web Services, which were modelled with regard to the business domain covered by SAP, are stored in SAP's Enterprise Service Repository. They have been analysed in a prior work [Zalt08] in order to generate a parameter ontology out of the input and output parameters of the respective services. This parameter ontology serves as a starting point for our research, leading us to suitable tools which we discovered during this research and which will help to fulfil the requirements discussed

⁷<http://technologies.kmi.open.ac.uk/aqualog/>

⁸A further discussion on AquaLog is given in section 4.1

⁹<http://gate.ac.uk/>

¹⁰<http://wordnet.princeton.edu/>

¹¹<http://www.ontotext.com/owlim/>. A thorough understanding is given in section 5.3.2

¹²<http://www.openrdf.org/>

in this section.

We also mention ideas which would be useful to pursuit, but are nevertheless out of scope of this work. Those features might however be of concern for a further enhancement of the results presented in this work.

- *Requirement 1*

A common problem regarding search engines in the domain of the Semantic Web is *knowledge overhead*, which is requiring users to be equipped with extensive knowledge of the back-end ontologies and knowledge bases or specific SQL-like querying languages in order to be able to formulate queries or to understand the search result. Hence, we want to *enable a search for service descriptions* - which were modelled with the help of ontologies - by *using natural language queries*. A user should not be expected to be knowledgeable of query languages in the semantic domain such as SPARQL¹³ or SeRQL (see section 4.2). The lower the barrier for a user to use this application right away, the higher its acceptance.

- *Requirement 2*

The search engine should be *portable to any ontology* describing Web Services and not be limited to one special formed ontology. The effort for customising an ontology to the search engine should be negligible, again raising the acceptance for this software system.

- *Requirement 3*

Sometimes, a question answering system might not be able to deliver meaningful results because of ambiguous data. Thus, if *no unambiguous search result* can be delivered, the *user's feedback* should be demanded. He should be able to encroach upon the search process and further specify his request.

- *Requirement 4*

The successful *results* of the search process should be *listed in a simple and clear way*. The user should not be required to have background knowledge of any kind in order to understand the search results.

- *Requirement 5*

The search engine needs to accept only queries formulated in the *English language* since the parameter ontology is modelled using English concepts and names. No multi-lingual support is necessary.

- *Requirement 6*

The tool should be easily accessible, thus, a graphical user interface (GUI) should be presented to insert one's queries.

There are several issues which would be for sure relevant for this work as well, but are not part of our requirements. However, the issues listed below can be understood as suggestions for further development of this application.

¹³<http://www.w3.org/TR/rdf-sparql-query/>

1. *Non-functional requirements*

It is not our goal to pursue non-functional requirements such as performance, security, Quality of Service or availability.

2. *Support for multi-lingual natural language queries*

As mentioned in Requirement 5, we focus on English as the only supported language to formulate queries. If this application faces high popularity sometime in the future, the need to support other languages will arise eventually.

3. *Ranking of search results*

Although the ranking of search results is an interesting feature to further involve the user into the search process, we will not include this in our work. We consider this as a feature not necessary to generally facilitate a search for Web Service descriptions modelled with semantic technologies.

The following section will outline the structure of our work and give an overview on the content of the different chapters.

1.3 Thesis Structure

The remainder of this work is outlined as follows:

Chapter 2 will introduce a variety of Web technologies which are currently in use and widely accepted. A basic understanding of these technologies is needed to comprehend the demonstrated drawbacks and follow the motivation for semantic technologies.

In *chapter 3* we will introduce the reader to the world of semantics and explain established semantic technologies. Some related work done in the fields of ontology-based discovery will be discussed as well.

Chapter 4 will start off with an overview of state-of-the-art semantic search tools, while identifying four categories of semantic search engines, according to the user interface they provide. Since we will focus on question answering (QA) tools, we will give a short presentation a QA-tool which we considered as suitable for our use case.

Chapter 5 will demonstrate our very own approach, showing how we design our solution by adjusting useful tools introduced in the previous chapter and by creating a so-called rule-set. This rule-set will be a key-element to make our solution even more portable as its underlying tools already are.

Our implementation part will be presented in *chapter 6*, where we discuss advantages as well as drawbacks of our question answering system.

Finally, we will summarise our results in *chapter 7* and draw some conclusions. An outlook highlighting further possibilities to improve and enhance our solution will be presented in *chapter 8*.

2

Current Web Technologies

In this chapter we will start by introducing the reader to the basic concepts of current Web technologies such as Web Services and the current industry standard of describing and advertising as well as discovering these services. Throughout this chapter we will highlight the drawbacks of current mechanisms and hence motivate the need for semantic enhancement, disposing the reader for established semantic technologies and methods introduced in the following chapter.

The primary purpose of the Web, as it was originally intended by its inventors, is to facilitate an easy way of sharing information between interested parties through Web pages, linked together via so-called hyperlinks. This infrastructure is what also facilitates the e-commerce applications for companies that want to reach their customers over the Internet. As foreshadowed in the introductory chapter, the concept of the Web has further developed and enhanced to not only include Web pages, but also *more active components*, meaning *Web Services*. Naresh et al. already state in [ApMe01], that "a *Web of services* is similar to the *Web of pages* in a lot of ways. For example, they will be connected to each other through some mechanism, and their users will be able to go from one service to the other using these links. However, the service Web is also very different from the Web of pages." The principal differences are in the form of service interactions, e.g. there is user-to-service interaction as well as service-to-service interaction.

The next section will take a closer look at Web Services, highlighting mechanisms which are used to describe Web Service interfaces as well as technologies enabling the registration and discovery of desired Web Services.

2.1 Web Services

Web Services are distributed and reusable software components programmatically accessible over standard internet protocols. They encapsulate a discrete functionality and *add a new level of functionality* on top of the current Web. The W3C explains in a working draft [W3C03] that "a Web Service is viewed as an abstract notion that must be implemented by a concrete agent. The agent is the concrete entity (a piece of software) that sends and receives messages, while the service is the abstract set of functionality that is provided" (e.g. booking of flight tickets in general and not a specific flight ticket). It further states: "To illustrate this distinction, you might implement a particular Web Service using one agent one day (perhaps written in one programming language), and a different agent the next day (perhaps written in a different programming language). Although the agent may have changed, the Web Service remains the same." This way, reusability, which is a desired concept in computer science, is provided.

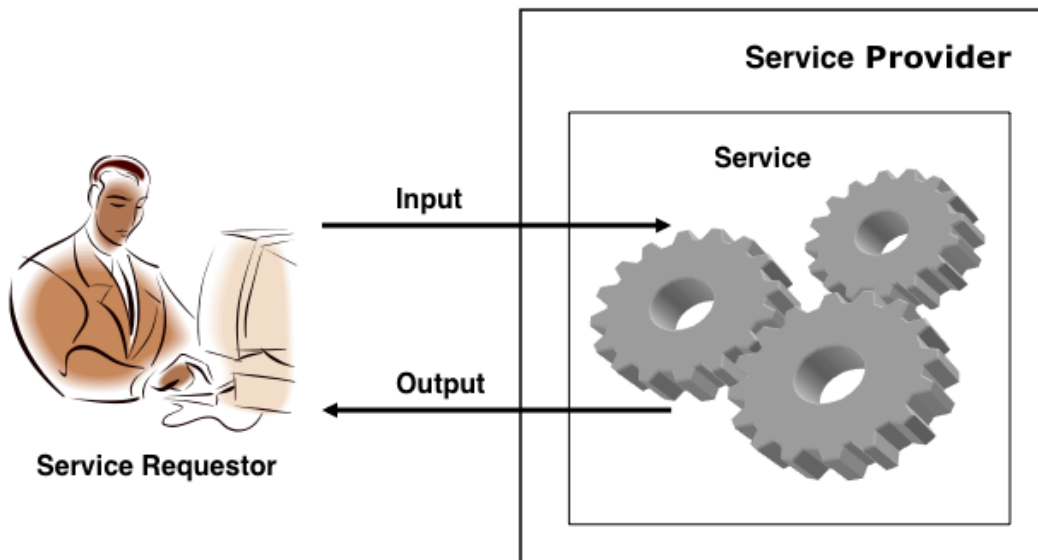


Figure 2.1: Service

In general, a *Service* is a software entity provided by a *Service Provider* (see figure 2.1). As explained in [Broe04], it performs an *action* - which is based on inputs - on behalf of a *Service Requester* and provides a result - the output - in return. A Web search engine is a good example to illustrate these concepts: when a user (service requester) types a query (input) into the search engine (service provider), the search engine tries to find relevant Web sites (action) and returns the findings to the user (output).

A Web Service can be described and characterised by his *properties*. Those properties can be subcategorised into so-called *functional* and *non-functional* properties. Concerning functional properties, we are speaking of e.g. the action it performs, which in- and outputs it supports and who performs the action. Non-functional

properties, on the other side, try to describe its qualities: Quality of Service (QoS), cost, performance and security can be specified here, for instance.

Messages are exchanged between Web Services and agents and Web Services. The mechanics of these message exchange are documented in a Web Service description (WSD). The WSD is, according to [W3C03], "a machine-processable specification of the Web Service's interface. It defines the message formats, datatypes, transport protocols, and transport serialisation formats that should be used between the requester agent and the provider agent. It also specifies one or more network locations ('endpoints') at which a provider agent can be invoked, and may provide some information about the message exchange pattern that is expected."

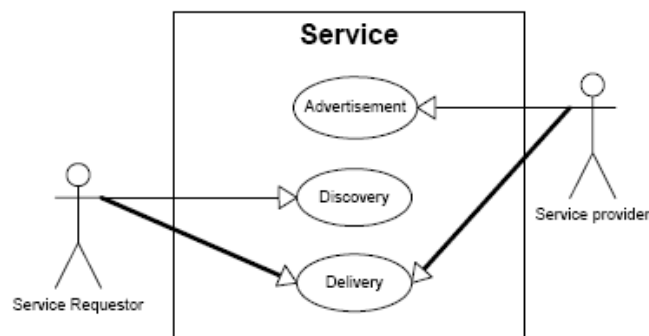


Figure 2.2: The service lifecycle, [O'EH02]

The operational lifecycle of a service consists of three successive phases: *Advertisement*, *Discovery* and *Delivery*. This is a generalisation of the model proposed in [O'EH02]. In the advertisement phase, the service provider creates a service description which is based on the properties of a service. This service is then enabled for use with the help of this description. The next step is on the user side, where the requester tries to find (i.e. via manual or automatic discovery) a service that satisfies his need in the discovery phase. It maybe the case that the service requester and provider are not associated with each other or even unaware of each other's existence. This is taken care of in the delivery phase, when both parties become associated with each other, which is demonstrated by the bold lines in figure 2.2.

The general structure of a Web Service as well as the technologies involved for describing, processing and communicating with Web Services are modelled as a stack, the so-called *Web Service Technology Stack*, which will be discussed in the next section.

2.2 The Web Service Technology Stack

The model of the Web Service Technology Stack was developed by the W3C Web Service Architecture Group and has been purposely designed on a very abstract level, meaning that no explicit technologies for the implementation have been specified in order not to limit the scope of Web Service technology. The official *Web Service*

Architecture Document is publicly available¹. The basic structure of the Web Service Technology Stack is depicted in figure 2.3.

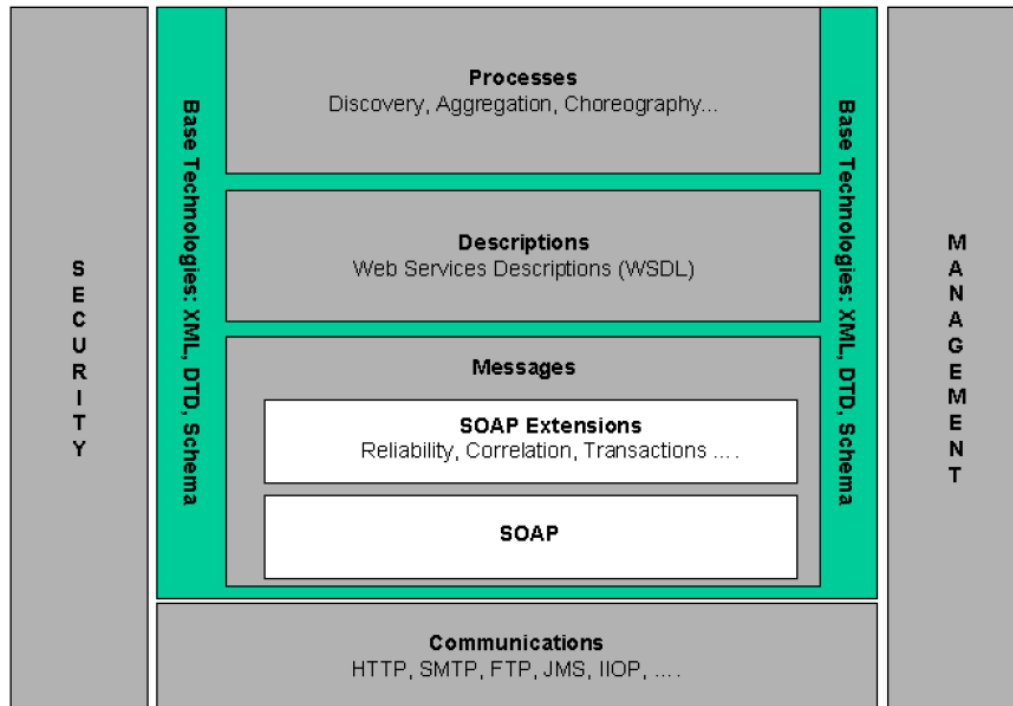


Figure 2.3: The Web Service Technology Stack, [StGA07]

The Web Service Technology Stack is very well explained in [StGA07]. At the very bottom of the stack we find the *Communications* block, which comprises generic transport mechanisms for sending messages over the Internet, e.g. HTTP, SMTP, FTP, and serves as the basis for all other layers. All these protocols provide specific benefits and drawbacks. Which one to use is not determined by the Web Service Technology Stack since the optimal choice may heavily depend on the use case.

The next layer includes the core technology of a Web Service. It starts with the *Messages* block, which provides basic functionality for encapsulating network messages in a way that makes them independent from any specific programming language or operating system. The message representation aims at being syntactically understood by humans and computers, as well as achieving syntactical interoperability in the Web Service world. This can be achieved with the general-purpose mark-up language XML², indicated by the green box around Messages, Descriptions and Processes in figure 2.3. The *Simple Object Access Protocol (SOAP)*³, being an XML language itself, provides a framework for packaging and exchanging XML messages, platform and application independent.

The layer defining technologies that are used for describing Web Services formally is called *Descriptions*. Such a description is of utmost importance because in order for a Web Service consumer to use a Web Service, he needs exact access parameters. Those parameters include data about the location of the service as well as a

¹<http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/>

²<http://www.w3.org/XML/>

³<http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>

specification of supported data types and operations. The most widely used and well established solution here is the *Web Service Description Language (WSDL, see section 2.3)*. It is important to note that WSDL does only cover the technical but not the semantical description of a Web Service.

On the very top of the stack we find the *Processes* block, which comprises several important processes in the Web Service world. For instance, when considering the combination of a number of services in order to complete a certain task, we talk about *Web Service aggregation* or *Web Service composition*. One of the most important processes in the field of Web Services is the *discovery* of a service, which we apply ourselves to in this work. It is the question of how we can locate a Web Service that fits our needs. The most popular solution here is Universal Description, Discovery and Integration (UDDI, see section 2.4).

There are certain issues that are relevant to all layers of the Web Service Technology Stack, for instance *Security*, which is located on the very left side of figure 2.3. In April 2002 Microsoft and IBM introduced the Web Service Security (WS-Security) specification. It provides a comprehensive security framework that is based on two other W3C standards as core components, namely XML Encryption and XML Signature.

Another very important issue, located on the right side of the figure, is *Management*. Since the Web Service technology primarily targets the domain of business applications, where the availability and reliability of a service might be crucial, it is very important that there are capable measures for monitoring and controlling the state of a Web Service. If we think of ‘pay per use’ scenarios, it is also desirable that the service provides a certain Quality of Service (QoS), for example by sending back query results within a given time interval. IBM addresses this issue in a framework called Web Service Level Agreement (WSLA), which has been introduced in 2003.

Web Services today face a couple of major problems. First of all, descriptions are syntactic and not enriched with semantic information, enabling machines to properly ‘understand’ what the Web Services do. This leads to the fact, that all tasks associated with Web Services application development have to be carried out by humans, such as discovery, composition and invocation, so scalability issues arise. The following section will focus on the “Descriptions” part of the stack, namely WSDL files.

2.3 WSDL

The Web Services Description Language (WSDL) is an XML-based language for describing Web Services as a set of network endpoints that operate on messages. According to [W3C01], “the operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint. Related concrete endpoints are combined into abstract endpoints (services).” This means that abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings.

This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service [W3C01].

Figure 2.4 illustrates the correlation between the above mentioned terms.

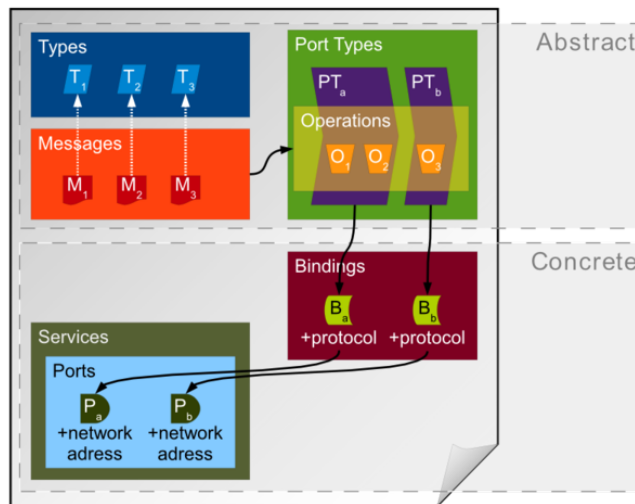


Figure 2.4: A representation of concepts defined by a WSDL 1.1 document, [<http://en.wikipedia.org/wiki/Image:Wsdl.png>]

In short, WSDL is used to define how Web Service interfaces look like. It is often used in combination with SOAP and XML Schema to provide Web Services over the Internet. A client program connecting to a Web Service can read the WSDL to determine what functions are available on the server. Any special datatypes used are embedded in the WSDL file in the form of XML Schema. The client can then use SOAP to actually call one of the functions listed in the WSDL.

The W3C Web Services Architecture Working Group does not prescribe a certain data format and therefore WSD (Web Service Description) documents might be written in any suitable language.

As [StGA07] states, it is important to note that WSDL does only cover the technical but not the semantical description of a service. Let us demonstrate this issue on the well-known *echo* operation. WSDL is able to express that a Web Service offers an operation *echo* which takes a string as parameter and returns another string. We as humans can guess from the operation's name that the two strings will be identical. But machines normally do only understand what they are explicitly told, thus they do need an additional semantic service description. This way they will be able to discover services that provide suitable operations for a given problem.

Mentioning the discovery process, we finally want to introduce the established registry *UDDI*, which enables businesses to publish service listings, discover each other and define how the services or software applications interact over the Internet.

2.4 UDDI

Web Services are meaningful only if potential users may find information sufficient to permit their execution. The focus of Universal Description Discovery and Integration (UDDI) is the definition of a platform-independent registry supporting the description and discovery of (1) businesses, organisations, and other Web Service providers, (2) the Web Services they make available, and (3) the technical interfaces which may be used to access those services [Comm]. Based on a common set of industry standards, including HTTP, XML, XML Schema, and SOAP, UDDI provides an interoperable, foundational infrastructure for a Web Services-based software environment for both publicly available services and services only exposed internally within an organisation. UDDI is an open industry initiative, sponsored by the global consortium OASIS (Organisation for the Advancement of Structured Information Standards, [OASI]).

This worldwide service registry can be seen as a huge phone book. A UDDI business registration consists of three components:

- *White Pages*
Address, contact, and known identifiers
- *Yellow Pages*
Industrial categorisations based on standard taxonomies such as NAICS (North American Industry Classification System)
- *Green Pages*
Technical information about services exposed by the business

One term not mentioned so far (and depicted in figure 2.5) is *SOAP*.

Originally defined as *Simple Object Access Protocol* [W3Ca], SOAP is a protocol specification for exchanging structured information in the implementation of Web Services in computer networks. It relies on Extensible Markup Language (XML) as its message format and usually relies on other Application Layer protocols, most notably Remote Procedure Call (RPC) and HTTP for message negotiation and transmission. Version 1.2 became a W3C Recommendation on 24 June 2003. The second and latest W3C Recommendation is from 27 April 2007.

However, UDDI has not prevailed in the domain of public Web Services for a number of reasons, according to [(see] and [ThSN03]:

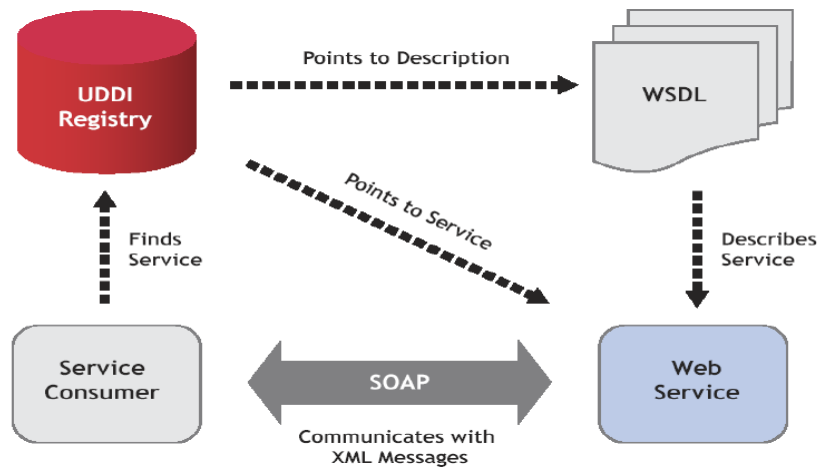


Figure 2.5: The Web Services Framework, [DoMa08]

- *UDDI is centred around programmatic access to the registry* and only a few mostly technically focused user interfaces are available. There is a considerable learning curve required for somebody not familiar with Web Services to understand the technicalities of UDDI specification. If we use an analogy to Web of pages, all what the new user of Web of pages requires to have is a pre-installed browser to start exploring websites. It is very important that somebody who learns for the first time about Web Service technologies (or services in general) can start exploring and enjoying them with a few simple steps.
- *There are no policies such as availability checks for Web Services* in place and thus many services in the public registries are outdated. A mechanism must be in place allowing to eliminate these services which are not available any more. Even in Google search results we sometimes find websites which are not working at a particular time, but in the long run, unavailable sites are getting always removed from Google's index.
- *There are no means for community feedback.* No service is perfect from the beginning. Practically there is only one possibility for feedback allowing the user to contact a provider using an email listed in the service description. But in the Web2.0 world it is not enough. While email might work for some particular cases, it does sustain a community around providers of Web Services, so in the long run the UDDI model has little applicability to the Web.
- *It is hard to expect that users would select a service based just on its WSDL definition and a short description.* You need pricing, terms and condition, service level agreements to name just a few. UDDI as API centric approach neglects that using a Web Service means first understanding it and making a contract - this requires to learn and understand various aspects of the service, not just its interface and short description.
- *The degree of automation for discovering Web Services is minimal.* UDDI does not represent service capabilities (metadata about services describing

operations, types and the bindings provided by a service), the information used only provides a tagging mechanism, and the search performed is only done by string matching on some fields they have defined. Thus, it is of no use for locating services on the basis of a semantic specification of their functionality. The lack of support of service selection based on non-functional attributes such as Quality of Service (QoS), security or availability is not negligible.

Although there are mechanisms in UDDI for realising concepts such as data replication, it is basically a centralised approach and therefore UDDI contradicts the concept of service distribution in some way.

Today, when the time of public UDDI registries is practically over (more about shut-down of IBM, Microsoft and SAP public UDDI registries in [Info05]), the process of publishing Web Services is mainly done by providing an access to the interface description as well as a couple of Web pages explaining the peculiarities of a service. This information can be then found by the crawlers of search engines.

Having stated the drawbacks of UDDI, it is clear that there must be found a more forward-looking technology to automate the way how Web Services are described, registered and discovered. The following chapter, which introduces to the world of semantics, will lead us to this forward-looking technology.

"The Semantic Web is not a separate Web but an extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation."

- Tim Berners-Lee

3

Step into the Semantic World

We now step into the world of semantic technology and guide the reader through its standards, technologies and concepts, while highlighting how they will help us to overcome the disadvantages mentioned in the previous chapters.

3.1 The Vision of the Semantic Web

In an article for the Scientific American in 2001 [BLHL01], Tim Berners-Lee, director of the World Wide Web Consortium, outlined his vision for the Semantic Web. There he states that the Web today - which can be assumed to be the biggest global database - lacks the existence of a semantic structure to keep the interdependency of its components, and as a result the information available on the Web is mostly human understandable. According to this article, the Semantic Web vision is one in which rich, *ontology-based* semantic markup becomes widely available, enabling sophisticated interoperability among agents, e.g., in the e-commerce area, and to support human web users in locating and making sense of information. Thus, the availability of semantic markup on the web opens the way to novel, sophisticated forms of *question answering*, which can potentially provide increased precision and recall (see definition of terms in section 3.3) compared to today's search engines. According to [GoGA08], a huge amount of data is conceptually related, but much of these relationships still have to be kept in human memory and are not stored in an understandable way for machines. The Semantic Web aims at machine-processable information. When the content is understood by machines, some assertions may come out of the content and new pieces of information will be produced. A very

suitable quote comes from [StGA07]: "The step from the current Web to the Semantic Web is the step from the manual to the automatic processing of information. This step is comparable to the step from the manual processing of goods to the machine processing of goods at the beginning of the industrial revolution. Hence, the Semantic Web can be seen as the dawn of the informational revolution, enabling automated intelligent services such as information brokers, search agents, information filters etc., and beyond that, further levels of software-system interoperability." Besides the technology and standards which already exist for syntactic representation of documents (e.g. HTML), there is also the need for standards representing the *semantic content*. Fortunately, a couple of such standards have been developed in the course of recent W3C standardisation efforts, as well as standards facilitating semantic *interoperability*, such as XML/XML Schema [BrPSM04], RDF/RDF Schema [BrGu04] (see section 3.7) and OWL [W3C04] (see section 3.8). The technology stack envisioned by the W3C is depicted in figure 3.1.

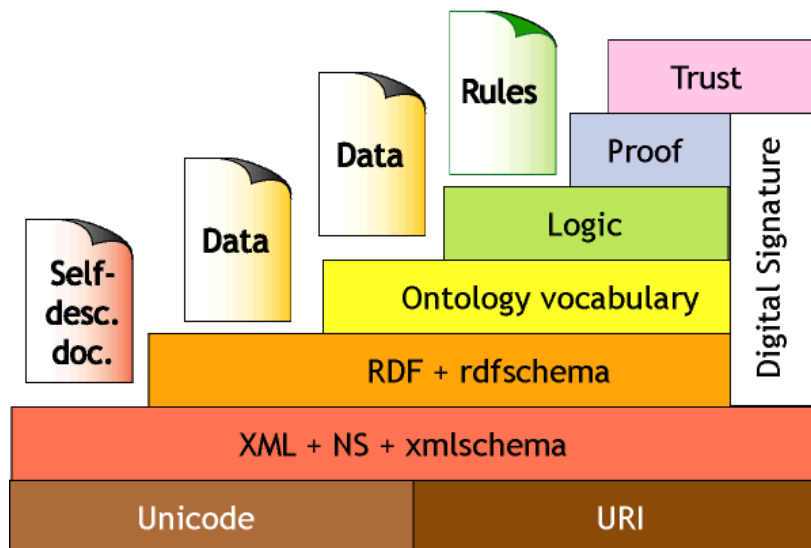


Figure 3.1: The Semantic Web technology stack, [Bern00]

The ground work for this Web layer cake is made up of the two blocks URIs and Unicode, since URIs are used in the semantic world to reference every modelled entity and make it uniquely identifiable. Above, we find XML as well as XML Schema, guaranteeing interoperability between technologies by making any semantic content serializeable. The third layer consists of RDF and RDFS (RDF schema), which is a language recommended by the W3C standardisation body for the representation of metadata about identifiable Web resources. The next layer is the ontology language. On top of the ontology language, there is a need for a language to express logic, so that information can be inferred and better put into relation. Once there is logic, it makes sense to use it to prove things. The proof layer enables everyone to write logic statements, and an agent can follow these semantic 'links' to construct proofs, so that validity of a statement, especially an inferred statement, can be checked. Trust is established by combining the proof layer with digital signature, indicated

by the white box on the right hand side of the picture. Consequently, ontology and ontology-based metadata are the basic ingredients for the Semantic Web layer cake.

In the previous chapter we introduced for the first time the concept of a Web Service. Having now arrived in the world of semantics, it is obvious that we will also talk about how Web Services and semantics can be combined. This is the topic of the next section.

3.2 Semantic Web Services

Semantic description of Web Services was proposed by the W3C in an attempt to resolve the heterogeneity at the level of Web Service specifications (including naming of parameters and a description of the service behaviour), and to take service oriented computing to a new level by allowing to semi-automate time-consuming programming tasks [ToGa07]. Using languages such as OWL¹, Web Services are unambiguously described by relating properties such as input and output parameters to common concepts, and by defining execution characteristics of the service. The concepts are defined in Web *ontologies*², which serve as the key mechanism to globally define and reference concepts.

As Küster et al. [Grad08] state: "At the core of SWS are solutions to the problem of SWS matchmaking, e.g., the problem of comparing a set of semantic service advertisements with a semantic request description to determine those services that are able to fulfil the given request."

The vision of the integration of Semantic Web Services in the range of syntactic vs. semantic as well as static vs. dynamic Web content is depicted in figure 3.2.

Before we take a closer look on already mentioned semantical description methods, we elaborate on the coherence of information retrieval and service discovery, further motivating the need for ontologies, RDF and OWL.

3.3 Information Retrieval and Service Discovery

Service discovery and information retrieval are related areas, according to [Broe04]. The information retrieval paradigm is about a user having a need for information, and a set of information objects from which this need has to be satisfied. Similarly, there is the service requester who has a need for a service, and a set of advertised services from which this need has to be satisfied. Therefore, some of the concepts of information retrieval can be applied to service discovery.

In information retrieval, there are two key quality measurements, which can also be applied in service discovery [Broe04]:

¹Further explained in section 3.8

²Further explained in section 3.5

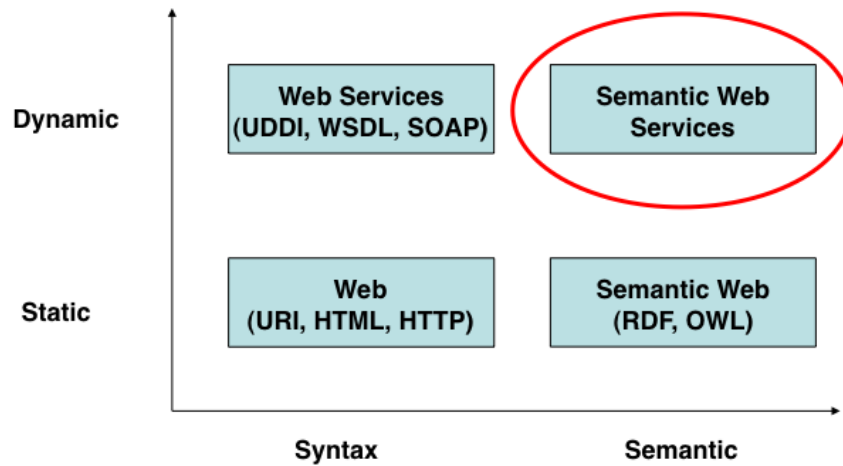


Figure 3.2: The vision of the Semantic Web, [DoMa08]

- Recall*

The number of relevant items (services) retrieved, divided by the total number of relevant items (services) in the collection. The highest value of recall is achieved when **all** relevant items (services) are retrieved.
- Precision*

The number of relevant items (services) retrieved, divided by the total number of items (services) retrieved. The highest value of precision is achieved when **only** relevant items (services) are retrieved.

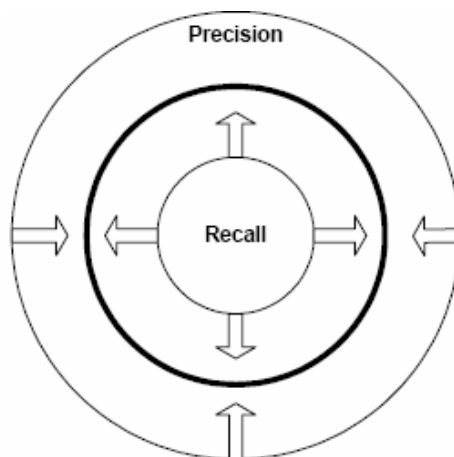


Figure 3.3: Recall and Precision, [Broe04]

To improve service discovery both recall and precision rates should improve. The circle with the bold line in figure 3.3 is the set with services of an optimal (100% recall, 100% precision) discovery result. Improved recall rates of a discovery method shift the inner border of the result more to the optimal border while improved precision rates shift the outer border more to the optimal border.

Currently, the keyword-based approach is the most widespread in industrial attempts to implement service retrieval. The most prominent example is the already mentioned UDDI protocol, which is an industry standard for locating Web Services through keyword and category search. The main drawback of UDDI, as well as other keyword-based approaches, is the lack of sufficient information for describing Web Services.

The next section will show, why a keyword-based search inevitably results in poor precision and recall.

3.4 Mechanisms of Service Discovery

Many of the existing service discovery mechanisms retrieve service descriptions that contain particular keywords from the user's query. In the majority of the cases, this leads to a low quality of retrieved results (and hence to low precision and recall) [Broe04]. There are two major reasons for this, namely *synonyms* and *homonyms*.

- *Synonyms*

Query keywords might be semantically similar but syntactically different from the terms in service description, e.g. 'buy' and 'purchase'.

- *Homonyms*

Query keywords might be syntactically equivalent but semantically different from the terms in the service description, e.g. 'order' in the sense of proper arrangement and 'order' in the sense of a commercial document used to request supply of something.

This ambiguity of keywords can lead to mismatches between the user query and service descriptions, which leads to a low quality discovery result.

Another problem with the *keyword-based* service discovery approaches is that they cannot completely capture the semantics of the user's query because they do not consider the relations between the keywords (e.g. if the query is 'order food', the relation between these keywords could indicate a need for a restaurant).

An approach to overcome these limitations is to use *ontology-based* (or *concept-based*) service discovery. In this approach, which we will follow in this work, ontologies are used for classification of the services based on their properties. This enables retrieval based on service types rather than keywords. Consider, for instance, a shop that defines that it sells 'music products' (e.g. CD's, DVD's). When the user specifies

that he wants to buy a 'CD', a syntactic mismatch occurs. When we use an ontology (that is shared by both parties) to derive that 'CD' is a type of 'music product', we can infer that this is a semantic match.

Furthermore, ontologies can specify interrelations among context entities and ensure common, unambiguous representation of these entities. Now, having mentioned so many times the concept of ontologies, it is time to take a closer look at what ontologies really are and how they are modelled.

3.5 Ontologies

After we have motivated the use of ontologies in service discovery, we will now introduce ontologies and explain their structure.

The Semantic Web relies heavily on the formal ontologies that structure its underlying data by providing vocabularies that can be used by applications in order to understand shared information. Ontology is a term borrowed from philosophy that refers to the science of describing the kinds of entities in the world and how they are related.

Studer et al. [StGA07] present a very compact definition of an ontology in the context of computer science:

An ontology is a formal explicit specification of a domain conceptualisation shared by the members of a community.

This statement captures several characteristics of an ontology, which is said to specify domain knowledge, namely the aspects of formality, explicitness, being shared, conceptuality and domain-specificity, which require some explanation. A very well explanation of these terms is given in [StGA07]:

- *formality*

An ontology is expressed in a knowledge representation language that provides a formal semantics (see section 3.7 and 3.8) . This ensures that the specification of domain knowledge in an ontology is machine-processable and is being interpreted in a well-defined way.

- *explicitness*

An ontology states knowledge explicitly to make it accessible for machines. Notions that are not explicitly included in the ontology are not part of the machine-interpretable conceptualisation it captures.

- *being shared*

An ontology reflects an agreement on a domain conceptualisation among people in a community. The larger the community, the more difficult it is to come to an agreement on sharing the same conceptualisation. Thus, an ontology is always limited to a particular group of people in a community, and its construction is associated with a social process of reaching consensus.

- *conceptuality*

An ontology specifies knowledge in a conceptual way in terms of symbols that represent concepts and their relations. The concepts and relations in an ontology can be intuitively grasped by humans, as they correspond to the elements in our mental model. Moreover, an ontology describes a conceptualisation in general terms and does not only capture a particular state of affairs. Instead of making statements about a specific situation involving particular individuals, an ontology tries to cover as many situations as possible that can potentially occur.

The basic elements of an ontology are *concepts*, *relations* and *instances*. Concepts represent the ontological categories that are relevant in the domain of interest. Concepts and instances are semantically related to each other using relations. Instances represent the named and identifiable concrete objects in the domain of interest, i.e. the particular individuals which are classified by concepts.

"An ontology can be viewed as a set of statements, expressed in terms of this vocabulary, which are also referred to as *axioms*. A simple axiom would, for example, state that 'Mister X is an employee', involving an instance and a concept. A more complex axiom could state that 'only employees of a particular company can be on trips booked by this company', imposing a restriction on a relation between two concepts", as stated in [StGA07].

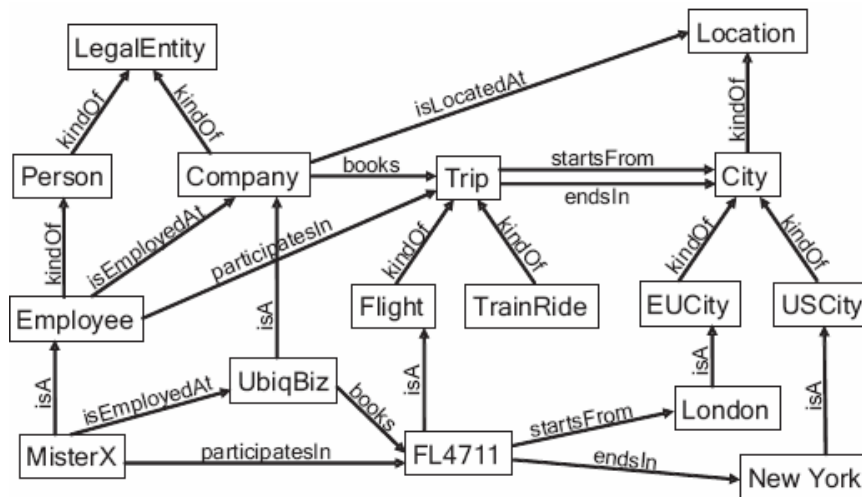


Figure 3.4: A graphical representation of a business trip ontology, [StGA07]

Figure 3.4 depicts a graphical representation of a business trip ontology and shows a graph whose nodes represent concepts and whose arcs represent relations between these concepts. The *business trips domain* contains typical concepts such as 'Company', 'Employee' or 'Flight', while typical relations would be 'books', 'isEmployedAt' or 'participatesIn'.

In summary, an ontology used in an information system is a conceptual yet executable and machine-interpretable model of an application domain. It can be used

by applications to base decisions on reasoning about domain knowledge. They create a *common understanding* on a specific domain (keyword-based service discovery drawbacks can be overcome, see section 3.4) and *enable semantic matchmaking*.

The next section deals with the concepts of ontologies and *knowledgebases*, since they are often used synonymously or without further distinction in texts dealing with this subject matter. However, we feel that we need to differentiate these two terms to not confuse the reader when both terms are used in the same context in this work.

3.6 Knowledge Bases vs. Ontologies

Sometimes, ontologies are confused with knowledge bases, in particular because the same languages (OWL, RDF, etc., see following sections for further explanation) and the same tools and infrastructure can be used both for creating ontologies and for creating knowledge bases. [HLMS08] draws a clear distinction: "Ontologies are the *vocabulary* and the formal specification of the vocabulary only, which can be *used for* expressing a knowledge base. It should be stressed that one initial motivation for ontologies was achieving interoperability between multiple knowledge bases. So, in practice, an ontology may specify the concepts 'man' and 'woman' and express that both are mutually exclusive - but the individuals Peter, Paul, and Marry are normally not part of the ontology." Consequently, not every OWL file is an ontology, since OWL files can also be used for representing a knowledge base.

This distinction is insofar difficult as individuals (instances) sometimes belong to the ontology and sometimes do not. Only those individuals that are part of the specification of the domain and not pure facts within that domain belong to the ontology. Sometimes it depends on the scope of the purpose of an ontology which individuals belong to it, and which are mere data. For example, the city of Innsbruck as an instance of the class 'city' would belong to a tourism ontology, but a particular train connection would not [HLMS08].

We suggest speaking of *ontological individuals* and *data individuals*. With ontological individuals we mean such that are part of the specification of a domain, and with data individuals, we mean such being part of a knowledge base within that domain.

3.7 RDF

Several ontology languages and variants with different expressiveness are available, one of which is the Resource Description Framework (RDF). It is a language recommended by the W3C standardisation body for the representation of metadata about identifiable Web resources, such as title and author of a Web page, topic and copyright information of an electronic document retrievable from the Web, or functionality and access conditions of a Web Service. The RDF metadata model

is based upon the idea of making statements about Web resources in the form of subject-predicate-object expressions, called *triples* in RDF terminology [HKRS08]. The subject denotes the resource, and the predicate denotes traits or aspects of the resource and expresses a relationship between the subject and the object. For example, one way to represent the notion 'The book Semantic Web is published by Springer' in RDF is as the triple: a subject denoting 'The book Semantic Web', a predicate denoting 'is published by', and an object denoting 'Springer'. Figure 3.5 illustrates this example.



Figure 3.5: A simple RDF graph to describe the relationship between a book and the publishing house Springer, [HKRS08]

RDF basically uses URIs (a generalisation of URLs - Uniform Resource Locators) to identify all resources. A collection of RDF statements intrinsically represents a labelled, directed RDF graph, whose nodes are resource URIs and whose arcs are properties. A node in an object position can be either a resource or an RDF literal, which represents a data value like the string 'mailto:someone@something.com' or some number. RDF, in general, is not limited to the description of Internet-based resources. A resource can be any kind of object: books, places, people, publishing houses, relationships between these things, abstract concepts and so on. These resources are obviously not retrievable and their URIs are exclusively used for identification.

Another Web-related aspect of RDF is its XML serialisation format in which RDF graphs are encoded for machine processing. The following example describes the RDF graph from figure 3.5 [HKRS08].

```

1 <?xml version='1.0' encoding='utf-8'?>
2   <rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns
3     /#' xmlns:ex='http://example.org/'>
4     <rdf:Description rdf:about='http://example.org/SemanticWeb'>
5       <ex:PublishedBy>
6         <rdf:Description rdf:about='http://springer.com/
7           PublishingHouse'>
8       </rdf:Description>
9     </ex:PublishedBy>
10  </rdf:Description>
11 </rdf:RDF>
  
```

Listing 3.1: Syntax example of an RDF file

Descriptions of resources are encoded using special XML tags from the RDF predefined vocabulary.

Recapitulatory, we can say that RDF puts the information in a formal way that a machine can understand. While XML only provides mechanisms to create structured data with elements and attributes, RDF also describes the data. The purpose of RDF is to provide an encoding and interpretation mechanism so that resources can be described in a way that particular software can *'understand'* it, meaning that software can access and use information that it otherwise could not use.

3.7.1 RDFS - RDF Schema

It is important to note that RDF is designed to provide a basic subject-predicate-object model for Web-data, it makes no data modelling commitments. In particular, no reserved terms are defined for further data modelling. As with XML, the RDF data model provides no mechanisms for declaring vocabulary that is to be used, according to [HKRS08].

RDF Schema - as an extension to RDF - is a mechanism that lets developers define a particular vocabulary for RDF data (such as the predicate `hasWritten`) and specify the kinds of objects to which predicates can be applied (such as the class `Writer`, [BrKH02]). RDFS does this by pre-specifying some terminology, such as `class`, `subClassOf` and `Property`, which can then be used in application-specific schemata. RDFS expressions are also valid RDF expressions, the only difference is that in RDFS an agreement is made on the **semantics** of certain terms and thus on the *interpretation* of certain statements. For example, the `subClassOf` property allows the developer to specify the hierarchical organisation of classes. Objects can be *declared to be instances* of these classes *using the type property*. Constraints on the use of properties can be specified using `domain` and `range` constructs.

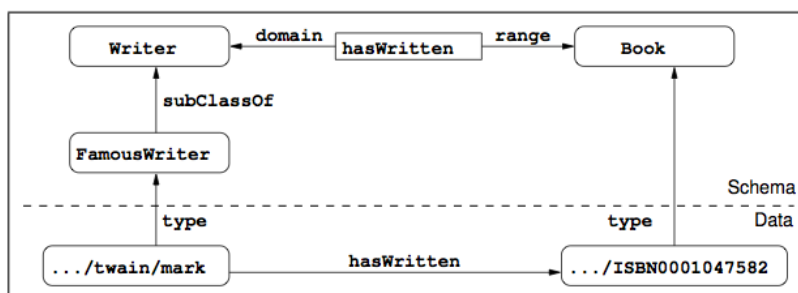


Figure 3.6: An example RDF Schema, [BrKH02]

Figure 3.6 tries to explain the aforementioned difference between Schema, depicted above the dotted line, and Data. We see an example RDF schema that defines vocabulary for an RDF example stating that Mark Twain wrote a book with a certain ISBN called 'The Adventures of Tom Sawyer': `Book`, `Writer` and `FamousWriter` are introduced as classes, and `hasWritten` is introduced as a property. A specific instance is described below the dotted line, in terms of this vocabulary.

The combined use of both RDF and RDFS is often referred to as RDF(S) and

provides a simple ontology language for conceptual modelling with some basic inferencing capabilities.

RDF's means of expression are restricted and sometimes not sufficient for the representation of complex interrelations. For that reason, a more expressive ontology language, called *OWL*, has been developed, which we will elaborate on in section 3.8. But first we want to mention a framework called *Sesame*, which provides for efficient storage and expressive querying of large quantities of metadata in RDF and RDF Schema. We will be using this framework later on in our own approach.

3.7.2 Sesame

Sesame is an open-source Java framework for efficient storage and expressive querying of large quantities of metadata in RDF and RDF Schema. It can be deployed on top of a variety of storage devices, such as relational databases, triple stores, or object-oriented databases, without having to change the query engine or other functional modules. Sesame offers support for concurrency control, independent export of RDF and RDFS information and a query engine for *RQL*, a query language for RDF that offers native support for RDF Schema semantics.

A central concept in the Sesame framework is the *repository*, which is a storage container for RDF. This can simply mean a Java object (or set of Java objects) in memory, or it can mean a relational database as mentioned above. Whatever way of storage is chosen, it is important to realise that almost every operation in Sesame happens with respect to a repository: when RDF data is added, it is added to a repository. When a query is executed, it is done with respect to a repository.

Sesame supports RDF Schema inferencing. This means that given a set of RDF and/or RDF Schema, *Sesame can find the implicit information in the data*. It supports this by simply adding all implicit information to the repository as well when data is being added. Note that inferencing in Sesame is associated with the type of repository that is used. Sesame supports several different types of repositories, of which some do support inferencing, others do not. Whether the user wants Sesame to do inferencing is a choice that depends very much on his application.

3.8 OWL

The Web Ontology Language OWL is a semantic markup language for publishing and sharing ontologies on the World Wide Web. OWL is developed as a vocabulary extension of RDF and has been standardised by the W3C [W3C04]. RDF is suitable for the modelling of simple ontologies and allows for the inferencing of implicit knowledge, but its restricted means of expression is not sufficient for the representation of complex interrelations.

An important issue for the design of OWL was the trade-off between expressivity of

the language on the one hand and scalability of reasoning on the other. To this end, OWL comes in three different flavours, namely OWL-Lite, OWL-DL (a fragment of Description Logics) and OWL-Full, reflecting different degrees of expressiveness. In our presentation of OWL we focus on OWL-DL as the most prominent language variant with the most support by the Semantic Web community. One of the key features of OWL-DL is that superclass-subclass relationships (subsumption relationships, useful for subsumption reasoning, see 3.10.1) in an ontology can be computed automatically by a reasoner.

Similar to RDF(S), OWL provides syntactic modelling constructs for the basic elements of an ontology, i.e. concepts, relations and instances. In OWL these are called *classes*, *properties* and *individuals*, respectively. In contrast to RDF(S), OWL-DL strictly separates classes from individuals and allows for building complex classes out of simpler ones by means of class *constructors*.

The named standards (RDF, RDFS and OWL) have a role to the Semantic Web similar to the role SQL played for the development and for the spread of the relational DBMS. Although primarily designed for use within the Semantic Web, the standards were widely accepted in areas like Enterprise Application Integration and life sciences.

3.9 Ontology-based Discovery Approaches

After having discussed the drawbacks of keyword-based search approaches in section 3.4 and introduced the term of ontologies as well as ontological markup languages, we will now give a short overview on interesting approaches taking into account ontologies and ubiquitous context information to increase recall and precision of search results.

Broens [Broe04] focuses on a semantic service discovery using ontologies to match service requests against service offerings and, thus, tries to avoid the major drawbacks of keyword-based service discovery. He also follows the idea of integrating contextual information of the user in the service request in order to regard the user's environment, therewith aiming at retrieving better-suited service results. Using the example of a hungry person travelling by car and searching for a place to eat, they consider contextual information like location (where is the nearest-by restaurant?), speed (which restaurant is the fastest to get to?), time (which restaurant is still opened?), personal interest (italian or fast food?) and car status (how much gas is still left in the tank?). This is an interesting approach, though it might not exactly meet the needs in our scenario where we imagine a SAP user looking for SAP Web Services located in the service repository.

Eran Toch et al. [ToGa07] base their approach on three strategies: (1) using the current research in semantic Web Services; (2) using approximation techniques to increase the recall of possible service compositions; (3) exploring indexing techniques

for sublinear response time. This approximate matching accounts for an amount of extra effort needed which is reflected in the ranking of results, combining both the semantic distance between query and result and the partiality of the result. A Web-based search engine, called OPOSSUM (Object-PrOcedure-SemanticS Unified Matching), was built that uses semantic methods for precise and efficient retrieval of Web Services, based on their WSDL descriptions³. OPOSSUM crawls the Web for WSDL descriptions, transforming them into ontological-based models of the Web Services. It does that by automatically augmenting the service properties with existing concepts, which are collected from ontologies on the Semantic Web. Solutions for issues of indexing, retrieval and ranking are provided within this framework. However, it would be desirable for the search engine interface to accept queries formulated in *natural language* (which is one of our major requirements as we will see in the next chapter) instead of keywords combined with parameters, such as "input:city output:flight booking", as it is the case in OPOSSUM.

Gopal et al. [GoGA08] pick up the idea of the Web-based search engine OPOSSUM but try to improve the user satisfaction and increasing the user trust on the search engine by improving the response time and reliability of Web Service repositories. This is achieved by a so called *Ontology based Service Index Annotator (OSIAN)* which is a module that can work in association with a search engine and acts as a mediator between the user and the search engine. OSIAN consists of two parts: (1) An availability checker and (2) RTub (Recent Tub). RTub is a tub of Web Services which have been recently checked for availability. There are two cases to be considered in the OSIAN message exchange: In case 1, RTub has the data required by the user and can directly supply the data to the user. The Web Services in the RTub need not be checked for availability, resulting in a quick reply by saving time for availability checking. In case 2, RTub does not have the data required by the user. Now the control is handed over to the search engine and the usual search procedure is carried out. The result is passed to OSIAN and it checks the availability of the Web Services in the result. Available Web Services get qualified to enter the RTub and to be displayed to the user.

Automatic availability checks for the Web Services in the repository without user intervention, provision of a cache effect when users access the services from the repository (making the response very fast) and reduction of the use of resources like processor time by reducing the number of disk access needed increase the overall performance of the search engine using OSIAN. These are central advantages of this approach.

Since this approach is based on the search engine OPOSSUM, it also does not accept queries formulated in natural language.

In [PKCH05], Jyotishman et al. propose a work that stresses the fact that different users may use different ontologies to specify the desired functionality and capabilities of a service, which rises the need for some kind of ontology mapping during service discovery, such that terms and concepts in the service requester's ontologies are

³A live view of this search engine can be found at <http://dori.technion.ac.il/>

brought into correspondence with the service provider's ontologies. This is done using so called interoperation constraints (ICs) between the terms and concepts of its ontologies to the domain ontologies. This is again an interesting approach, however, these ICs are defined manually which might be quite time consuming and thus calls for a more (semi-)automatic solution. Furthermore, they propose a taxonomy for the non-functional attributes, namely QoS, which provide a better model for capturing various domain-dependent and domain-independent QoS attributes of the services. This way they are facing the problem that different aspects of QoS might be important in different applications and different classes of Web Services (e.g. *bits per second* may be an important QoS criterion for a service providing online streaming multimedia, whereas *security* is more important for a service which provides online banking).

3.10 Reasoning

We already used the term *reasoning* a couple of times in this chapter without providing a definition, which we will make up for now.

Studer et al. [StGA07] give a very comprehensive explanation, stating that the way in which we, as humans, process knowledge is by reasoning, i.e. the process of reaching conclusions. Similarly, a computer processes the knowledge stored in a knowledge base by drawing conclusions from it, i.e by deriving new statements that follow from the given ones. The basic operations a knowledge-based system can perform on its knowledge base are typically denoted by *tell* and *ask* [RuNo95]. The tell-operation adds a new statement to the knowledge base, whereas the ask-operation is used to query what is known. The statements that have been added to a knowledge base via the tell-operation constitute the explicit knowledge a system has about the domain of interest. The ability to process explicit knowledge computationally allows a knowledge-based system to reason over a domain of interest by deriving implicit knowledge that follows from what has been told explicitly.

This leads to the notion of logical consequence or *entailment*. A knowledge base KB is said to entail a statement α if α 'follows' from the knowledge stored in KB. A knowledge base entails all the statements that have been added via the tell-operation *plus* those that are their logical consequences.

The inference procedures implemented in computational reasoners aim at realising the entailment relation between logical statements [RuNo95]. They derive implicit statements from a given knowledge base or check whether a particular statement is entailed by a knowledge base.

Studer et al. [StGA07] further state that an inference procedure that only derives entailed statements is called *sound*. Soundness is a desirable feature of an inference procedure, since an unsound inference procedure would potentially draw wrong conclusions. If an inference procedure is able to derive every statement that is entailed by a knowledge base then it is called *complete*. Completeness is a desirable property

as well, since a complex chain of conclusions might break down if only a single statement in it is missing. Hence, for reasoning in knowledge-based systems we desire sound and complete inference procedures.

3.10.1 Subsumption Reasoning

Subsumption corresponds with checking whether a class description subsumes (is more general than) another class description. By doing this for all classes in the knowledge base, one can compute the subsumption hierarchy. By checking the place in the subsumption hierarchy of a given class description, this class description is classified with respect to the knowledge base and hidden relationships with other classes in the knowledge base become visible [StGA07].

Querying for subsumption between two classes underlies the most important usage of reasoning in the OWL language, namely *classification*. The following OWL ontology allows for the automatic classification of two classes that are not explicitly put in subsumption relation.

```

1 {
2   class(SplitCity complete
3     intersectionOf(City
4       restriction(governedBy minCardinality(2))))),
5   class(GreekTurkishCity partial
6     intersectionOf(City
7       restriction(governedBy someValuesFrom(oneOf(Greece)))
8       restriction(governedBy someValuesFrom(oneOf(Turkey))))),
9   DifferentIndividuals(Greece Turkey)
10 }
```

Listing 3.2: Ontology example demonstrating subsumption, [StGA07]

The first statement introduces split cities, while the second statement introduces a class *GreekTurkishCity* for cities which are governed by both Greece and Turkey. The third statement assures the two involved countries to be distinct. From the knowledge specified in the ontology, *GreekTurkishCity* is a subclass of *SplitCity* and a DL reasoner would derive the statement *subClassOf(GreekTurkishCity SplitCity)* as a logical consequence. By checking subsumption between all the named classes in an OWL ontology, an inferred class hierarchy can be established.

In this thesis, we will focus on subsumption reasoning as a technique to locate the desired services in a service discovery process. We will see later on in section 5.1.2 how exactly subsumption reasoning will be applied in our approach.

Up to this point, we have introduced the reader to the current standard of Web technologies, have highlighted their drawbacks and consequently motivated the need for semantic technology. After this step into the semantic world, we now reached the point to introduce question answering techniques, which are to be introduced in the following chapter.

But before we move to the next chapter, we shortly want to discuss the topic of ontologies design. We give a quick digest on current approaches to guidelines on how to create rich and rigorous ontologies. This is a subject matter which will occupy us later on in section 6.7.

3.11 Quality of Ontology Design

There is no official standard or industry-proof solution set up in the Semantic Web community on how to design a 'correct' or perfectly modelled ontology. However, there are approaches to guidelines in this direction, e.g. the so-called *ontology design patterns* (ODPs). ODPs are ready made modelling solutions for creating and maintaining ontologies. They intend to help in creating rich and rigorous ontologies with less effort. These patterns are analogous to the programming design patterns used by software engineers.

First of all, there is the *Ontology Engineering and Patterns Task Force (OEP)* [En(O)], established by the *Semantic Web Best Practices and Deployment Working Group* of the W3C. The aim of this task force is "to provide guidance for developers of Semantic Web applications. In particular, we focus on the engineering of semantic web ontologies, through the publication of notes that document common and reusable ontology patterns, and general ontology engineering best practices."

Furthermore, there has been a tutorial offered by Rector et al. [RSNK⁺04] at the ISWC2004 (the 3rd International Semantic Web Conference), called *Ontology Design Patterns and Problems: Practical Ontology Engineering using Protege-OWL*. The outline of this tutorial states: "This tutorial will introduce attendees to the concept of ontology patterns and discuss key patterns developed by the Task Force and others. Example patterns include quantities and units, value partitions, n-ary relations, problems of distinguishing classes and individuals, the choice between representing relationships as classes or properties, and qualified cardinality restrictions."

Another online resource dedicated to the topic of ontology design patterns is *Ontology Design Patterns.org (ODP)* [ODP]. This is a semantic web portal dedicated to ontology design patterns (OPs) for the Semantic Web developed in the context of the NEON⁴ project. They offer amongst others a public catalog of ODPs "focused on the biological knowledge domain. ODPs in this catalog have been collected elsewhere or created 'in house' and they are open for discussion." They categorise their ODPs in *Extension ODPs* (such as Nary_DataType_Relationship⁵, Exception, and Nary_Relationship), *Good Practice ODPs* (e.g. Entity_Feature_Value⁶, Selector, and Normalisation), and *Domain Modelling ODPs* (e.g. Interactor_Role_Interaction⁷ and Sequence).

⁴<http://www.neon-project.org>

⁵http://www.gong.manchester.ac.uk/odp/html/Nary_DataType_Relationship.html

⁶http://www.gong.manchester.ac.uk/odp/html/Entity_Feature_Value.html

⁷http://www.gong.manchester.ac.uk/odp/html/Interactor_Role_Interaction.html

Another related topic concerning the quality of ontology design is about the *metrics* of ontologies. Vrandečić et al. [VrSu07] discuss in their paper the need for proper metrics which allow the fast and simple assessment of an ontology by taking into account not only the structural metrics of an ontology but the semantics of the ontology language as well. They state that measuring ontologies is necessary to evaluate ontologies both during engineering and application, thus, performing quality assurance and controlling the process of improvement. They introduce two key properties of metrics, namely *ontology normalisation* and *stability of metrics*. The former is a preprocessing step in order to align structural measures with intended semantic measures by explicating some features of the semantics of an ontology within its structure, so that the structural metrics actually capture the semantics they are supposed to capture. The latter examines how stable the metrics are with regards to the open world assumption⁸ of OWL DL ontologies.

⁸In formal logic, the open world assumption is the assumption that the truth-value of a statement is independent of whether or not it is known by any single observer or agent to be true. It is the opposite of the closed world assumption which holds that any statement that is not known to be true is false.

4

Question Answering

The previous chapter guided the reader through the world of semantics and emphasised the need for semantic technologies.

Semantic search promises to produce precise answers to user queries by taking advantage of the availability of explicit semantics of information which is to be queried. Furthermore, the underlying semantic relations of metadata can be exploited to support the retrieval of information which is closely related to the query term. Lei et al. [LeUM06] present in their paper an overview of state-of-the-art semantic search tools, while *identifying four categories of semantic search engines*, according to the user interface they provide. They show that, although these tools do enhance the performance of traditional search technologies, some of them are not suitable for naive users. With naive users we mean ordinary end users who are not necessarily familiar with domain specific semantic data, the concept of ontologies, or SQL-Like query languages such as SPARQL [W3Cb] or SeRQL (see section 4.2 for further explanation).

The mentioned categories are:

- *Form-based search engines*

These are search engines, which provide sophisticated web forms that allow users to specify queries by choosing ontologies as well as their contained classes, properties, values, etc. One example mentioned in [LeUM06] for this category is the SHOE search engine¹. Although, taking semantic information into account, SHOE is only suitable for those users who are familiar with the underlying ontologies and knowledge bases. Naive users, as defined above, have difficulties in understanding these forms and formulating queries.

¹<http://www.cs.umd.edu/projects/plus/SHOE/search/>

- *RDF-based querying languages fronted search engines*

Such search engines, e.g. the Corese search engine², usually provide a sophisticated querying language to support semantic data querying. But again, considering naive end users, this is not an approach promising a wide diffusion since the user would have to be familiar with both the back-end ontologies and the provided querying language.

- *Semantic-based keyword search engines*

According to [LeUM06], the search process of such search engines often comprises two major steps: (1) finding an instance match for the user keyword and (2) retrieving instances which are closely related to the instance match of the user keyword. Although, it takes into account semantic information without requiring the user to have any knowledge about ontologies or query languages, it is still a keyword-based approach and does not allow for complex queries.

- *Question answering tools*

The fourth and last category summarises examples of ontology-based question answering engines, such as AquaLog [LUMP07] and ORAKEL [Cimi04]. Such search engines make use of natural language processing technologies to reformulate natural language queries into either ontological triples (which is the case in AquaLog) or into specific query languages (as in ORAKEL). These tools appear to be ideal for naive users. Lei [LeUM06] however notes that the search performance is heavily influenced by the performance of the used natural language processing techniques.

In order to avoid the linguistic processing and potential loss in performance, Lei et al. [LeUM06] advocate a keyword-based search over a natural language question answering. A major goal of their work is to hide the complexity of semantic search from end users and to make it easy to use and effective for naive users. A Google-like query interfaces is provided where a user can type in his query in form of keywords. We, however, will from now on focus on a question answering machine which supports queries formulated in natural language, in order to meet our first requirement. After having conducted intensive research, we decided to utilise AquaLog. Our motivation for this decision will be explained in the following section.

4.1 AquaLog

AquaLog allows users who are knowledgeable about a certain domain and have a question in mind to query the semantic markup which is viewed as a knowledge base (KB) [LUMP07]. The aim is to provide a system which does not require users to learn specialised vocabulary, or to know the structure of the knowledge base, but they have to have some idea of the contents of the domain. So, AquaLog is designed

²<http://www-sop.inria.fr/acacia/soft/corese/>

to serve as an *interface for the Semantic Web*.

It is a portable question-answering system which takes queries expressed in natural language and an ontology as input, and returns answers drawn from one or more KBs, which instantiate the input ontology with domain-specific information, according to [LUMP07]. It is portable in the sense that the AquaLog system allows the user to choose an ontology and then ask queries with respect to the universe of discourse covered by the ontology (essentially a semantic viewpoint, so to speak). The configuration time required to customise the system for a particular ontology is negligible. This being said, it seems that AquaLog is also a perfect tool to meet our requirements 1 and 2, as established in the introductory chapter.

Thus, AquaLog seems to be predestined especially as a front-end to organisational semantic intranets where an organisational ontology is used as the basics for semantic markup, which is *exactly our case*.

AquaLog presents an elegant solution in which different strategies are combined together in a novel way. It makes use of the GATE NLP platform (see section 4.1.2.1), string metric algorithms, WordNet (see section 4.1.2.2) and a novel ontology-based *Relation Similarity Service* to make sense of user queries with respect to the target KB by applying *syntactic and semantic analysis of the question*. Moreover, it also includes a learning component to obtain domain-dependent knowledge by creating a lexicon. This ensures that the performance of the system improves over time, in response to the particular community jargon used by end users.

Finally, AquaLog makes use of a generic plug-in mechanism, which means it can be easily interfaced to different ontology servers and knowledge representation platforms. One of the plug-ins used in this work is the so-called *SesamePlugin* which has been developed to collaborate with Sesame, an open source RDF framework with support for RDF Schema inferencing and querying. We elaborated on Sesame in section 3.7.2.

To give a first impression on how AquaLog works and what it is capable of, we will have a closer look at an illustrative example.

4.1.1 An Illustrative Example

In [LUMP07], a quick illustrative example is given which best illustrates at a coarse-grained level how AquaLog actually works. Since the development of AquaLog took place at the Knowledge Media Institute and Centre for Research in Computing (The Open University)³, some exemplary questions are formulated with a focus on the KMI domain (research institute, researchers, projects, staff of the institute, etc.).

The AquaLog architecture can be characterised as a cascaded model, according to [LUMP07], in which a NL query gets translated by the *Linguistic Component* into a set of intermediate triple-based representations, which are referred to as the *query-triples*. Subsequent to this process, the *Relation Similarity Service (RSS)* component

³<http://kmi.open.ac.uk/>

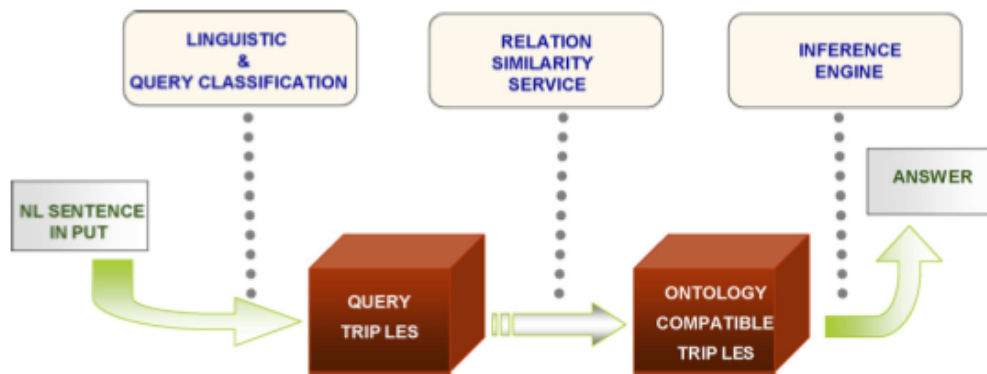


Figure 4.1: The AquaLog data model, [LUMP07]

takes as an input these query-triples and further processes them to produce the ontology-compliant queries, called onto-triples, as shown in figure 4.1.

The decision of transforming the NL query into a triple-format is based on the fact that knowledge representation formalisms for the semantic web, such as RDF or OWL, also subscribe to this binary relational model and express statements as $\langle \text{subject}, \text{predicate}, \text{object} \rangle$.

The exemplary ontology of the KMI is modelled quite well in order to illustrate the power of AquaLog since many statements are expressed in the $\langle \text{subject}, \text{predicate}, \text{object} \rangle$ form, where *predicate* forms a relation between the subject and the object. At first, to demonstrate the possibilities of AquaLog, we will adduct some queries which targets the mentioned KMI ontology. The interested reader is encouraged to try some of these queries on his own by using the AquaLog demo available online⁴. Later on, when describing our own approach, we will certainly consider the applicability of our SAP Enterprise Services ontology, but since some problems might occur with the SAP ontology, we will elaborate on this particular issue at a later stage in section 6.7.

In the context of the academic domain in the KMI department, an exemplary question given by [LUMP07] is "what is the homepage of peter who has an interest on the semantic web?". As we can see, this question actually consists of two parts, namely the *main* question about a person's homepage, and a *modifier* part, which imposes a constraint on the person to look for ("who has an interest ..."). This NL query is translated by AquaLog into the ontology-compliant logical query

- $\langle \text{what is?}, \text{has-web-address}, \text{peter-scott} \rangle$ and
- $\langle \text{person?}, \text{has-research-interest}, \text{Semantic Web area} \rangle$

expressed as a conjunction of triples containing variables. In order to transform the NL query to these triples, some steps have been invoked before.

⁴<http://plainmoor.open.ac.uk:8080/aqualog2/index.html>

Focusing on the first part ("what is the homepage of peter?"), the Linguistic Component created the intermediate query-triple <what is?, homepage, peter>. The role of the RSS is now to map the intermediate form into the target ontology-compliant query. If ambiguities occur which the RSS can not resolve with the information available, it calls the user to participate in the question answering (QA) process. Figure 4.2 demonstrates on the left side how this disambiguation step is achieved with the help of the user. By using string metrics, the system is unable to disambiguate between Peter-Scott, Peter-Sharpe, Peter-Rutherford, etc. Moreover, the user feedback is required to disambiguate the term "homepage" since it is the first time AquaLog came across this term. No synonyms have been found using WordNet, and the ontology does not provide further information to disambiguate. After the user has chosen "has-web-address", he can activate the *Learn* checkbox so that the system is able to learn the user's vocabulary and context for future occasions.

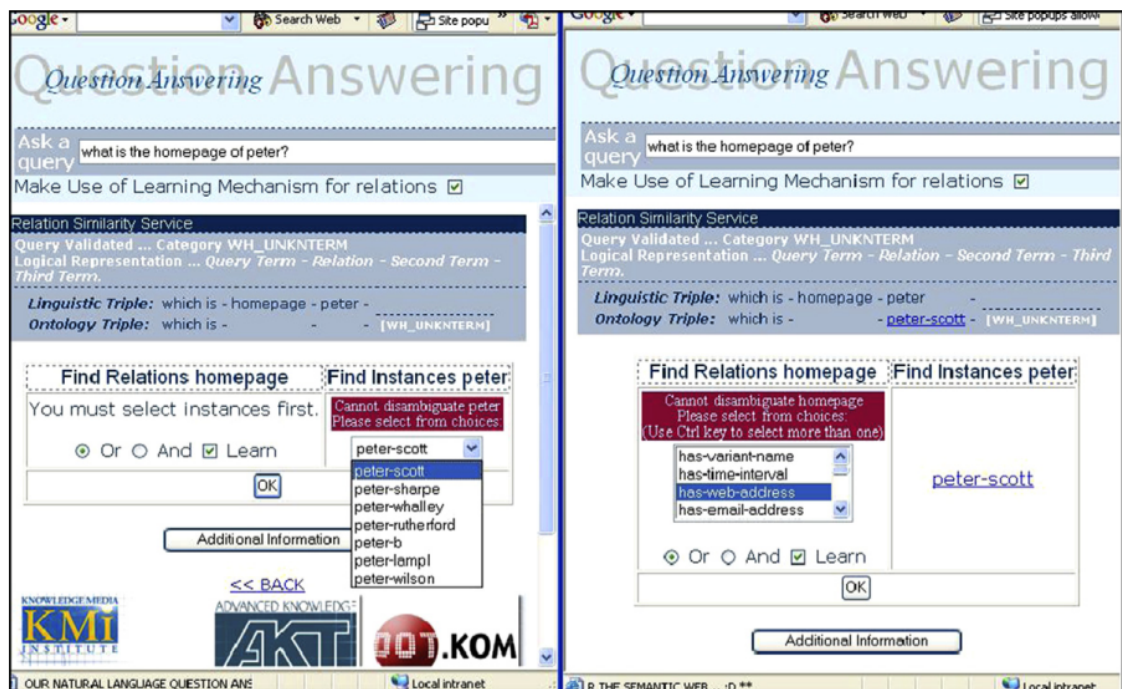


Figure 4.2: Illustrative example of user interactivity to disambiguate a basic query, [LUMP07]

In figure 4.3 we repeat the aforementioned question, but this time with the modifier "who has an interest on the semantic web". This time, AquaLog does not need any assistance from the user, given that, by analysing the ontology, only one of the "Peters" has an interest in Semantic Web, and only one possible ontology relation, "has-research-interest" (taking into account taxonomy inheritance) exists between "person" and the concept "research area", of which "semantic web" is an instance. Since the similarity relation between "homepage" and "has-web-address" has been learned before by the learning mechanism, the performance improves over time. We can now see that a major challenge for AquaLog is to efficiently *deal with complex queries*, which could include more than just one term, but two or even more. As pointed out in the above exemplary query, these terms may take the form of modifiers

that change the meaning of other parts of the query, and they can be mapped to instances, classes, values, or combinations of them, in compliance with the ontology to which they subscribe.

The screenshot shows the AquaLog Question Answering interface. At the top, there's a search bar with the query "which is the homepage of peter who has an interest on the semantic web". Below the search bar, there's a section titled "Relation Similarity Service" which displays the query validation and logical representation. The query is validated as "Category PATTERNS_2" and its logical representation is "Query Term - Relation - Second Term - Third Term". The interface shows the linguistic triple and ontology triple for the query. The linguistic triple is "which is - homepage - peter" and the ontology triple is "which is - has-web-address - peter-scott". The interface also shows the mapping of the query term "peter" to "has-research-interest" and "semantic-web-area". The answer to the question is "peter-scott has-research-interest semantic-web-area". The value of "has-web-address" for "peter-scott" is "news.kmi.open.ac.uk/peterblog".

Question Answering

Ask a query [Examples](#) You are logged as anonymous

Make Use of Learning Mechanism for relations ☒

Relation Similarity Service

Query Validated ... Category PATTERNS_2

Logical Representation ... Query Term - Relation - Second Term - Third Term

Linguistic Triple: which is - homepage - peter -

Ontology Triple: which is - has-web-address - peter-scott - [WH_UNKINTERM]

Note: The Lexicon (learning mechanism) is mapping to { has-web-address }

person - has-research-interest - semantic-web-area - [WH_GENERICTERM]

Note: Cannot find a relation to map. The only possible relation is has-research-interest

The answer to the question:

peter-scott has-research-interest semantic-web-area

The value of has-web-address for peter-scott is/are:

[<< BACK](#)

KNOWLEDGE MEDIA **KMI** INSTITUTE

ADVANCED KNOWLEDGE **AKT**

Powered by **KOM**

Figure 4.3: Illustrative example of AquaLog disambiguation, [LUMP07]

4.1.2 AquaLog's Helpers

Up to now, we have tried to give the reader a basic understanding of the functioning of AquaLog. We now go on and present the technologies on which AquaLog relies and which we have mentioned before, namely GATE and WordNet.

4.1.2.1 GATE

GATE, a General Architecture for Text Engineering [GATE], is a leading toolkit for text mining and was first released in 1996, then completely re-designed, re-written, and re-released in 2002. The system is now one of the most widely-used systems of its type and is a relatively comprehensive infrastructure for language processing software development. It is comprised of an architecture, a free open-source framework (or SDK) and graphical development environment and used for all sorts of language processing tasks, including Information Extraction in many languages. Information Extraction (IE) is a process which takes unseen texts as input and produces fixed-format, unambiguous data as output. This data may be used directly for display to users, to be stored in a database or spreadsheet for later analysis, or may be

used for indexing purposes in Information Retrieval (IR) applications. IE covers a family of applications including named entity recognition, relation extraction (which is amongst others the use case in AquaLog) and event detection.

4.1.2.2 WordNet

WordNet is a large lexical database for the English language, developed at the University of Princeton. Nouns, verbs, adjectives and adverbs are grouped into sets of cognitive synonyms (synsets), each expressing a distinct concept. Synsets are interlinked by means of conceptual-semantic and lexical relations. Numerous lexical semantic relations are formally modelled, like *hyponyms* (subsumption from a more-general term) and *antonyms* (negation, a term with the opposite meaning). WordNet is freely and publicly available for download [Word]. Its structure makes it a useful tool for computational linguistics and natural language processing, as it is our case in this work.

4.2 SeRQL

At the beginning of this chapter, we shortly mentioned an SQL-like query language named *SeRQL*. SeRQL will be an important aspect in our own approach, thus, it should be introduced to the reader, giving a basic overview and understanding.

SeRQL [BrKa04] stands for Sesame RDF Query Language and is a querying and transformation language loosely based on several existing languages, most notably RQL, RDQL and N3 (see [HBEV04] for further explanation on and a comparison between RQL, RDQL and N3). As the name suggests, it is the query language used in the Sesame framework which we introduced in section 3.7.2. Its primary design goals are unification of best practices from query languages and delivering a light-weight yet expressive query language for RDF that addresses practical concerns.

SeRQL supports generalised path expressions, boolean constraints and optional matching, as well as two basic filters: select-from-where and construct-from-where. The first returns the familiar variable-binding/table result, the second returns a matching (optionally transformed) subgraph.

During this work we had to change some SeRQL-queries built-in in the AquaLog code, specifically in the class *SesamePlugin.java*. We will comment this issue in chapter 6 (see section 6.4). For a detailed description of the syntax of SeRQL, please refer to the online documentation⁵.

⁵<http://openrdf.org/doc/sesame/users/userguide.html#chapter-serql>

"Design is not just what it looks like and feels like. Design is how it works."

- Steve Jobs

5

Design and Adaptation

This chapter presents the overall architecture of our approach and contribution. The previous chapters have already introduced some software frameworks and tools which are utilised to meet our requirements. We now explain how all of these components interact and seamlessly integrate.

5.1 The Global Architecture

At first, we shortly want to mention Tomcat, the overall runtime environment in which our application runs. Apache Tomcat is a servlet container developed by the Apache Software Foundation (ASF)¹ and implements the Java Servlet and the JavaServer Pages (JSP) specifications from Sun Microsystems. Tomcat provides an environment for execution of Java-code on web servers. It offers a "pure Java" HTTP web server.

Figure 5.1 illustrates all the components involved in our application.

When we were first talking about AquaLog in chapter 4, we stated that our aim is to provide a system which does not require users to learn specialised vocabulary, or to know the structure of the knowledge base, but they have to have some idea of the contents of the domain. Hence, we want a system which is designed to serve as an *interface for the Semantic Web*, giving the end user the possibility to formulate questions in natural language. This possibility is provided via the *query interface* of our architecture.

¹<http://tomcat.apache.org/>

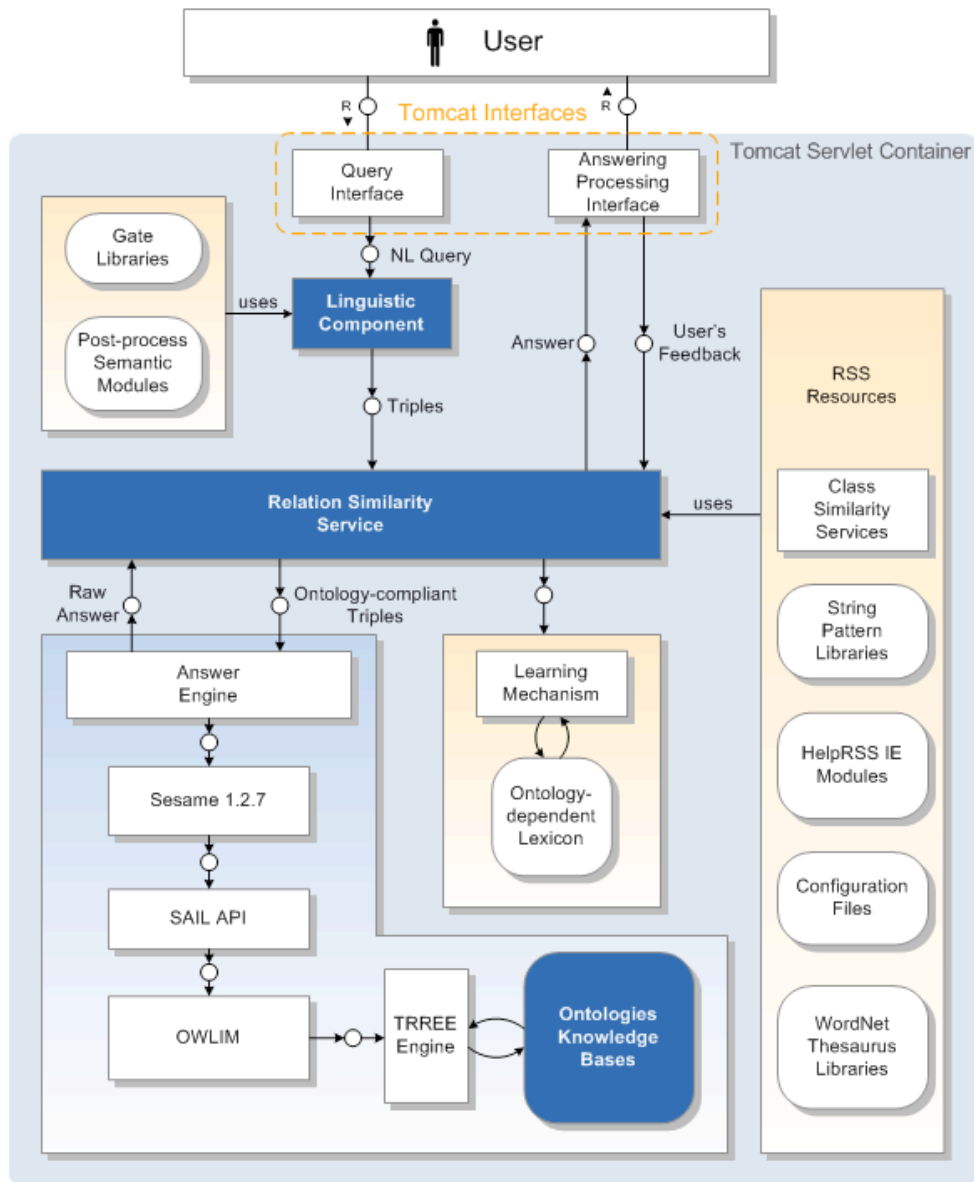


Figure 5.1: The overall architecture of our approach

Two of the major components of this system are the *Linguistic Component* and the *Relation Similarity Service* (RSS), both of them portable and independent. They are actually part of the AquaLog system [LUMP07], as most of the components in the diagram. Our own approach and contribution is depicted by the L-Box at the lower left corner. We will elaborate on this a little later. The Linguistic Component and the RSS will be addressed in a separate section (see section 5.1.1 and 5.1.2) as well.

Since AquaLog is modular, it makes use of a plug-in mechanism, which allows it to be configured for different Knowledge Representation (KR) languages, such as OCML (Operational Conceptual Modeling Language)², RDF³ and OWL⁴.

AquaLog also caches basic indexing data from the target *Knowledge Bases* (KBs) at initialisation time to reduce the number of calls/requests to the target KB and to guarantee real-time question answering.

At startup, some initialisation information is read from the *configuration files*, which require human intervention for adapting the system to a new domain, thus realising portability. In section 6.6, we will show how to adjust these configuration files for our use case.

Lopez et al. [LUMP07] state that a query can be translated into one or more *linguistic-triples*⁵ (see 'Triples' between Linguistic Component and Relation Similarity Service in the diagram), of which each of them can be translated into one or more *ontology-compliant-triples*.

Moreover, this architecture also includes a *learning component* to obtain domain-dependent knowledge by creating a *lexicon*. This ensures that the performance of the system improves over time, in response to the particular community jargon used by end users and retrieved via the *user's feedback*.

As soon as an answer to the user's question has been generated, it is delivered to the user via the *answering processing interface*.

We will now take a closer look at the Linguistic Component and the RSS, as well as the resources they utilise in order to fulfil their tasks.

5.1.1 From Questions to Query Triples

The process of translating from natural language (NL) to the triple format, which is used to query the ontology, is done by the *Linguistic Component*.

One of the resources used by this component is the GATE infrastructure (see section 4.1.2.1) which communicates with the Linguistic Component through the standard

²<http://technologies.kmi.open.ac.uk/ocml/>

³Earlier explained in section 3.7

⁴Earlier explained in section 3.8

⁵See next section and section 5.3 for more on the triple approach.

GATE API. The GATE processing resources, e.g. *English tokenizer*, *sentence splitter*, *POS (Part-Of-Speech) tagger* and *VP (Verb Phrase) chunker*, sequentially invoked by the Linguistic Component, return a set of syntactic annotations associated with the input query, as stated in [LUMP07]. These annotations include information about sentences, tokens, nouns and verbs. As an example, we get voice and tense for the verbs and categories for the nouns, such as determinant, singular/plural, conjunction, possessive, determiner, preposition, existential, wh-determiner etc., and information about which is the main verb (the one that separates the nominal group and the predicate).

The team developing AquaLog extended the set of annotations returned by GATE by identifying noun terms, relations, question indicators (which/who/when, refer to section 6.6 to see how this contributes to portability) and patterns of *types of questions*. This could be achieved through the use of GATE *JAPE transducers*, a set of JAPE grammars that they wrote for AquaLog. JAPE⁶ is an expressive, regular expression based rule language offered by GATE. Examples of the annotation obtained after the use of these JAPE grammars can be seen in figures 5.2 and 5.3. Currently, the Linguistic Component dynamically identifies around 14 different question types or intermediate representations, which can be viewed in detail in Appendix A of [LUMP07]. We will not further elaborate on the syntax of JAPE grammars and how to adapt them to our application since this goes beyond the scope of our work. However, it needs to be stated that this is a very powerful tool and should be taken into account when further developing this application. The interested reader is therefore referred to section 4.2 of [LUMP07].

Basically, at a coarse-grained level, there are three main groups of queries (based on the number of triples needed to generate an equivalent representation of the query). We think that it is essential for the user to be aware of the different kinds of queries he or she can ask our question answering engine, so we will shortly go into detail.

- Basic queries
- Basic queries with clauses
- Combinations of queries

Basic queries can be further divided into basic queries requiring an affirmation/negation (e.g. "is vanessa working as a research fellow?", see figure 5.2) or a description as an answer. Moreover, there is the big set of queries constituted by a *wh-question* (starting with: what, who, when, where), for instance "are there any phd students in dotkom?", where the relation is implicit or unknown, or "which is the job title of John", where no information about the type of answer expected is provided. Also imperative commands such as list, give, tell, and name are treated as wh-queries.

Basic queries with clauses are those three-term queries which have modifiers included. Considering the request "list all the projects in the knowledge media institute about the semantic web", there are two modifiers, namely "in knowledge media

⁶<http://gate.ac.uk/sale/tao/index.html>

institute” and ”about semantic web”. These modify the meaning of the other syntactic constituents (”projects”). The main challenge here is to identify the constituent to which each modifier has to be attached. The Relation Similarity Service (RSS) is responsible for resolving this ambiguity through the use of the ontology, or in an interactive way by asking for user feedback. The Linguistic Component’s task is, therefore, to pass the ambiguity problem to the RSS through the intermediate representation.

At last, we have the case where a query can be a *composition of two basic queries*. There are three ways in which queries can be combined. Firstly, by using a ”and” or ”or” conjunction parameter, secondly by making use of modifiers as in ”which researchers wrote publications related to social aspects” (where the second clause modifies one of the previous terms), and finally by combining two basic patterns, e.g. ”are there any planet news written by researchers working in akt?”, where the two linguistic triples will be <planet new, written, researchers> and <which are, working, akt>.

Once the intermediate representation is created, prepositions and auxiliary words, such as ”in”, ”about”, ”of”, ”at”, ”by”, ”does”, are removed from the query-triples because in the current implementation their semantics are not exploited to provide any additional information.

The resultant query-triple is the input for the Relation Similarity Service that processes the combinations of queries as explained in section 5.1.2.

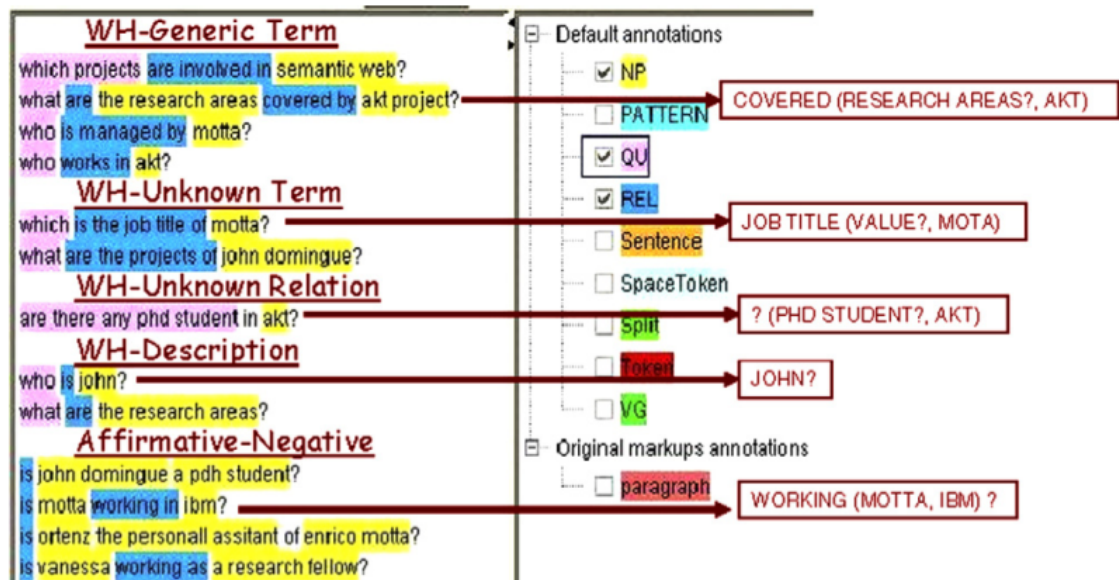


Figure 5.2: Example of GATE annotations and linguistic triples for basic queries, [LUMP07]

It is important to emphasise that, at this stage, all the terms are still strings or arrays of strings, without any correspondence with the ontology. This is because the *analysis is completely domain independent* and is entirely based on the GATE analysis of the English language.

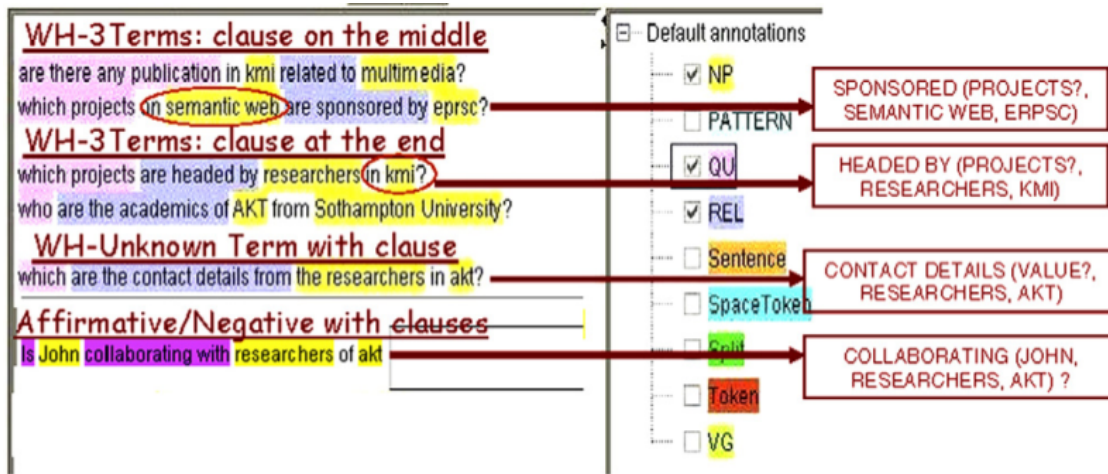


Figure 5.3: Example of GATE annotations and linguistic triples for basic queries with clauses, [LUMP07]

A key feature for each triple is its *category*.

Depending on the category, the triple tells us how to deal with its elements, what inference process is required and what kind of *answer* can be expected. These categories are i.e. called *wh-generic term* (e.g. "Who are the researchers in the semantic web research area?"), *wh-unknown term* (like "Show me the job title of Peter Scott") or *wh-unknown relation* (like "Are there any projects about semantic web?"). For a detailed list of all possible categories identified by AquaLog, please refer to Appendix A of [LUMP07].

Let us take for example the query "what are the research areas covered by the akt project?". Thanks to the category, the Linguistic Component knows that it has to create one triple that represents an explicit relationship between an explicit generic term and a second term. It gets the annotations for the *query terms*, *relations* and *nouns*, preprocesses them and generates the following query-triple: <research areas, covered, akt project> in which "research areas" has correctly been identified as the query term (instead of "what").

As soon as the query-triples are generated, they are forwarded to the Relation Similarity Service for further processing.

5.1.2 From Triples to Answers

The RSS is the backbone of the question-answering system and responsible for generating an *ontology-compliant logical query* after the NL query has been transformed into a term-relation form and classified into the appropriate category. By using *string similarity matching*, generic lexical resources such as *WordNet* (see section 4.1.2.2), a *domain-dependent (or ontology-dependent) lexicon* which is obtained through the use of a *Learning Mechanism*, and by looking at the structure of the *ontology* and the information stored in the KBs, the RSS tries to make sense of the input query (delivered in the form of query-triples). An important aspect of the RSS is that it

is interactive in the sense that the user will be required to interpret the query when ambiguity arises between two or more possible terms or relations. This is a very important aspect of our system, because we can therefore fulfil the third requirement, as introduced at the beginning of this thesis.

Relations and concept names are identified and *mapped* within the ontology through the RSS.

Proper names, instead, are normally mapped into instances by means of string metric algorithms. However, this can be a disadvantage since some questions may not be accepted without some facilities for dealing with unrecognised terms. As an example, take the question "is there any researcher called Thompson?". This query would not be accepted if Thompson is not identified as an instance in the current KB.

However, a partial solution is implemented for affirmative/negative types of questions, where a query contains more than one instance of which one is recognised, e.g. in the query "is Enrico working in ibm?", where "Enrico" is mapped into "Enrico-motta" in the KB, but "ibm" is not found. The answer in this case is an indirect answer, namely the place where Enrico Motta is working.

Some sentences are structurally and syntactically ambiguous and although general world knowledge does not resolve this ambiguity, within a specific domain it may happen that only one of the interpretations is possible.

To give the reader an impression on how RSS works for *basic queries*, we will start off with a simple question, for instance "what are the research areas covered by akt?" (see figure 5.4).

The query is classified by the Linguistic Component as a basic generic-type (one *wh-query* represented by a triple formed by an explicit binary relationship between two explicit terms). The first step for the RSS is to identify that "research areas" is actually a "research area" in the target KB and "akt" is a "project" through the use of string distance metrics and WordNet synonyms if needed. As soon as a successful match has been found, the challenge arises to find a relation which links "research areas" or any of its subclasses to "projects" or any of its superclasses, which can be achieved via the inheritance of relations through the subsumption hierarchy (remember section 3.10.1, where we stated that we want to focus on subsumption reasoning). By analysing the taxonomy and the relationships in the target KB, the RSS finds out that the only relations between both terms are "addresses-generic-area-of-interest" and "uses-resource". With the help of the WordNet lexicon, the RSS is able to identify the word "addresses" as a synonym for "covered", and therefore suggests to the user that the question could be interpreted in terms of the relation "addresses-generic-area-of-interest".

Whenever multiple relations are possible candidates for interpreting the query, if the ontology does not provide ways to further discriminate between them, string matching is used to determine the most likely candidate, using the relation name, the Learning Mechanism, eventual aliases, or synonyms provided by lexical resources

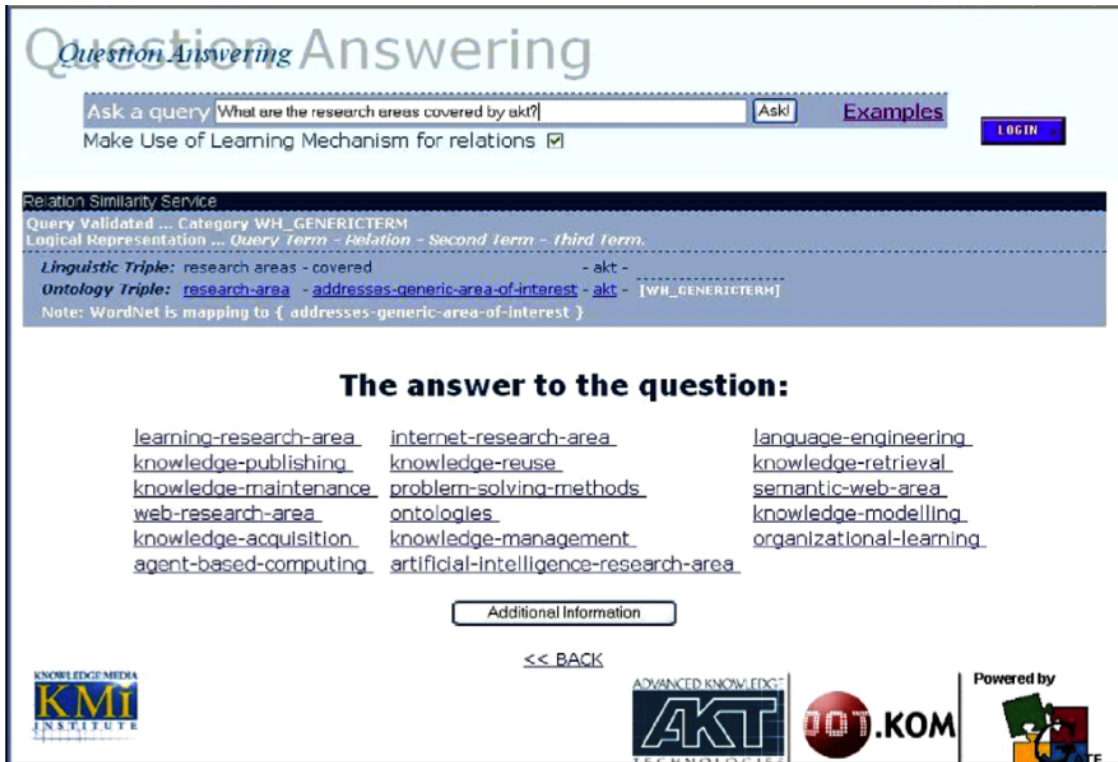


Figure 5.4: Example of AquaLog in action for basic generic-type queries, [LUMP07]

such as WordNet. If no relations are found by using these methods, then the user is asked to choose from the current list of candidates.

In many cases, the RSS has to cope with situations in which the structure of the intermediate query does not match the way the information is represented in the ontology. For instance, the query "who is the secretary in Knowledge Media Institute?", as shown in figure 5.5, may be parsed into $\langle \text{person}, \text{secretary}, \text{kmi} \rangle$. While this is the result of purely following linguistic criteria, it does not really match the way the ontology is organised, namely in terms of $\langle \text{secretary}, \text{works-in-unit}, \text{kmi} \rangle$. This is where the RSS shows its intelligent reasoning power, as it is able to reason about the mismatch, re-classify the intermediate query and generate the correct logical query. The procedure is as follows. The Linguistic Component classifies the above question as a "basic generic type". The first step is now to identify that KMI is a "research institute", which is also part of an "organisation" in the target KB, and to realise that *who* could be pointing to a "person"⁷. So now, the problem becomes (similarly to the previous example) one of finding a relation which links a "person" (or any of its subclasses like "academic", "students", "professors", ...) to a "research institute", but in this particular case, there is also a successful matching for "secretary" in the KB, in which *"secretary" is not a relation but a subclass of "person"*. Following this clue, the triple is classified from a generic one formed by a binary relation "secretary" between the terms "person" and "research institute" to a generic one in which the relation is unknown or implicit between the terms "secretary" (which is more specific than "person") and "research institute".

⁷See further explanation in section 6.6.



Figure 5.5: Example of RSS in action for relations formed by a concept, [LUMP07]

We could also go on and elaborate on how RSS procedures work for basic queries with clauses and combinations of queries, but this would somewhat go beyond the actual scope of this work. The example shown above had the aim to demonstrate the capabilities and power of the RSS component and give an idea on how this very important component basically works. Therefore, the interested reader is referred to [LUMP07] for continuative comprehension.

Before we go on and discuss our own contribution by explaining the integration of the Sesame framework and OWLIM plug-in into the AquaLog architecture, we first want to mention some limitations concerning the question answering abilities, and, thereby, motivate the need for this very useful integration.

5.2 Query Answering Limitations

During their evaluation, Lopez et al. [LUMP07] identified several failures which they divided in different categories, since a query may fail at several different levels. We think that it is also important to mention these categories in order to better understand the causes of a potential failure.

- *Linguistic failure*

This occurs when the NLP component is unable to generate the intermediate representation (but the question can usually be reformulated and answered).

- *Data model failure*

This occurs when the NL query is simply too complicated for the intermediate representation.

- *RSS failure*

This occurs when the Relation Similarity Service is unable to map an intermediate representation to the correct ontology-compliant logical expression.

- *Conceptual failure*

This occurs when the ontology does not cover the query, e.g. lack of an appropriate ontology relation or term to map with, or if the ontology is wrongly populated in a way that hampers the mapping (e.g., instances that should be classes).

- *Service failure*

In the context of the Semantic Web, they believe that these failures have less to do with shortcomings of the ontology than with the lack of appropriate services, defined over the ontology (like aggregation services for understanding key words like "most", "best five", etc.).

A user evaluation, conducted by Lopez et al. [LUMP07], showed that the linguistic failures were by far the most common problems. Errors were amongst others due to questions not recognised or classified (e.g. when a multiple relation is used, as in "what are the challenges *and* objectives [...]") or questions annotated wrongly (such as tokens not recognised as a verb by GATE, e.g. "*present* results") and therefore relations annotated wrongly.

In many cases questions can be easily reformulated by the user to avoid these failures so that the question can be recognised by the NLP component (linguistic failure), avoiding the use of nominal compounds (typical RSS failure) or avoiding unnecessary functional words, for instance different, main, and most of (service failure).

AquaLog claims to be portable in the sense that it is ontology-independent, but as we will see now, there are major restrictions which are based on assumptions made by AquaLog about the design of an ontology. These restrictions deter AquaLog from being able to handle ontologies modelled in a certain way, which is relatively wide-spread on the semantic Web.

5.3 The Triple Approach Problem

The components of AquaLog operate best under the assumption that all relationships are modelled in the triple approach <subject, predicate, object>, with predicate being the relation.

There exist a number of ontologies which are well-known in the semantic community and tend to serve as standard ontologies when introducing a user not yet familiar with the design and assembly of an ontology to the world of semantics and OWL files. When dealing with ontologies for the first time, it is likely that one will come across the wide-spread "pizza ontology" or "wine ontology", for instance. In fact, a tutorial for introducing an interested person to ontologies with the help of the free,

open-source ontology editor Protégé⁸ has been provided by the CO-ODE project [HKRS⁺04]. This tutorial uses the pizza ontology to demonstrate how ontologies are modelled step by step.

One of the first things we tried after exploring the AquaLog system and its capabilities was to formulate a number of simple questions in natural language and let AquaLog answer it with respect to this well-known and wide-spread pizza ontology. Since we knew that we would have to redesign the given parameter ontology [Zalt08] because of its lack of proper relationship design (no object properties were available), we first wanted to make sure that AquaLog is able to deal with a quite common, well-designed ontology. Figure 5.6 depicts the structure of the pizza ontology.

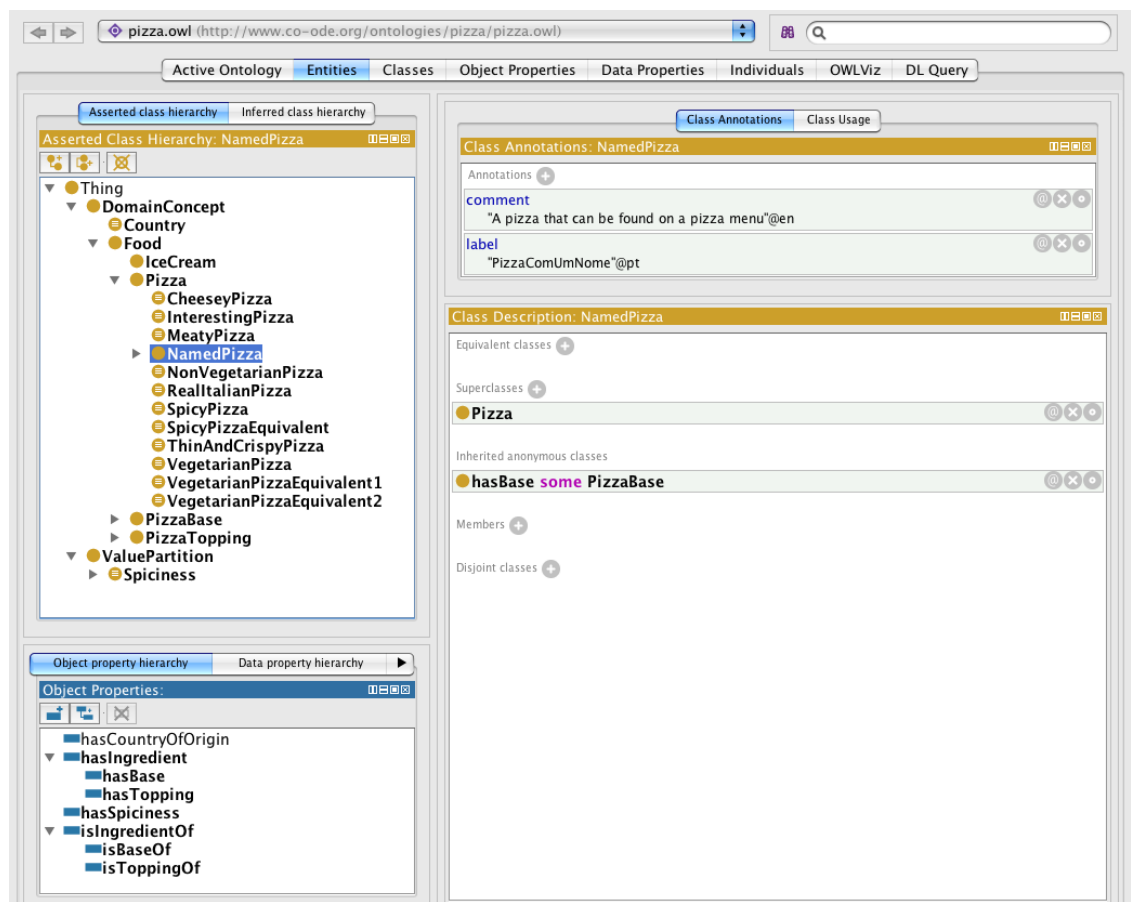


Figure 5.6: The structure of the pizza ontology as illustrated by Protégé

On the left side, the hierarchies of the asserted classes as well as the object properties are visualised, while on the right side we see the class description of the marked class, listing Equivalent classes, Superclasses, Members, etc.

Beneath the branch "NamedPizza" we find a number of popular pizza names such as FruttiDiMare, QuattroFormaggi or Margherita. Figure 5.7 shows how the Margherita pizza is modelled.

⁸<http://protege.stanford.edu/>

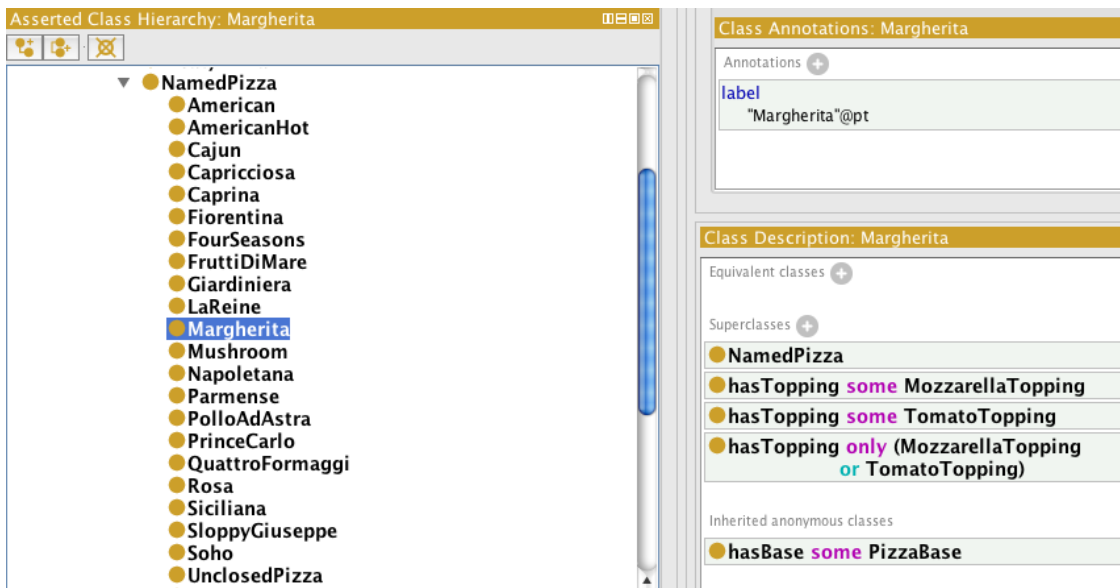
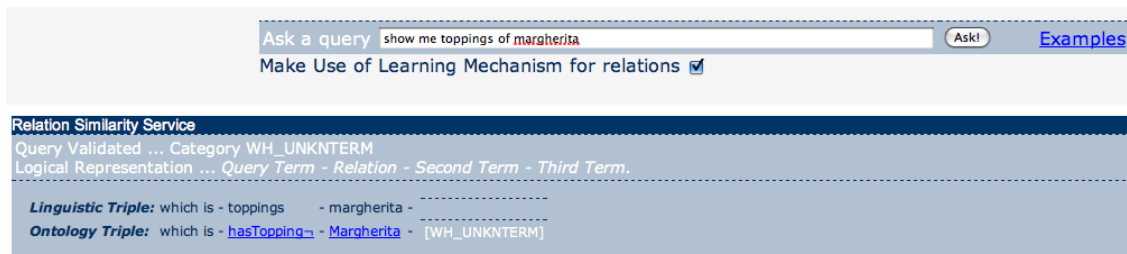


Figure 5.7: Margherita pizza modelled in OWL

One of the first questions which came into our mind was "show me toppings of margherita", expecting a list of all toppings - namely TomatoTopping and MozzarellaTopping - for the Margherita pizza. After the RSS component of AquaLog was successfully able to recognise that "toppings" corresponds to the relation (object property) "hasTopping" and that "margherita" matches the class "Margherita" in the "NamedPizza" branch, it unfortunately fails to list the expected toppings of the Margherita pizza as shown in figure 5.8. The system states that there are *no values for class Margherita using relation hasTopping*.



The answer to the question:

The are not values for class [Margherita](#) using relation [hasTopping](#)

Figure 5.8: AquaLog is unable to satisfactorily answer a simple question for pizza toppings

We then realised, that the term 'Margherita' is used as an object, and not as a subject. If we take a closer look at the ontology triple (see figure 5.8), we see that AquaLog generated the triple <which is, hasTopping, Margherita>, assuming that Margherita is the topping. Hence, we reformulated our question and asked "what is the topping of margherita", hoping that AquaLog would take into account the object property *isToppingOf* (which is the inverse property of hasTopping). This question

resulted in an error message of the Relation Similarity Service, stating: "Cannot validate query: possible malformed expression. In sentence *what is the topping of margherita* Missing a relation in the pattern

The query: what

The nouns: margherita

The relations: is topping

The pattern: margherita

This seems to be a typical linguistic failure, as explained in section 5.2. However, skipping the definite article "the" ("what is topping of margherita") lead us to the screen where we could choose between two relations, namely *hasTopping* and *isToppingOf*, as illustrated in figure 5.9.

The screenshot shows the AquaLog web interface. At the top, there is a search bar with the query "what is topping of margherita" and a button labeled "Ask!". Below the search bar, there is a checkbox labeled "Make Use of Learning Mechanism for relations" which is checked. The main content area is titled "Relation Similarity Service" and displays the query "Query Validated ... Category WH_UNKNTERM" and its logical representation "Logical Representation ... Query Term - Relation - Second Term - Third Term". Below this, it shows the "Linguistic Triple: which is - topping - margherita -" and the "Ontology Triple: which is - Margherita - [WH_UNKNTERM]". A dialog box titled "Find Relations topping" is open, displaying a message: "Cannot disambiguate topping. Please select from choices: (Use Ctrl key to select more than one)". It lists two options: "http://www.co-ode.org/ontologies/pizza/pizza.owl#isToppingOf" and "http://www.co-ode.org/ontologies/pizza/pizza.owl#hasTopping". Below the list are radio buttons for "Or", "And", and "Learn", with "Learn" selected. An "OK" button is at the bottom of the dialog. To the right of the dialog, there is a section titled "Find Instances margherita" which displays the instance "Margherita".

Figure 5.9: AquaLog asks for user feedback to disambiguate a relation term

Unfortunately, this reformulation of the query did not help either, we received the same response, telling us that there are *no values for class Margherita using relation isToppingOf*.

It seems odd at first that AquaLog can not find the explicitly stated relations between Margherita and its toppings as indicated in figure 5.7. It took us a substantial amount of time to figure out why exactly this is the case and what we can do about it.

This is where SwiftOWLIM comes into play. Before we explain why exactly we use SwiftOWLIM here, we have to introduce a "feature" of RDF called *Blank Nodes* and explain why this conflicts with the way AquaLog understands ontologies.

5.3.1 Blank Nodes

Referring to the W3C⁹, a blank node "is a node that is not a URI reference or a literal. In the RDF abstract syntax, a blank node is just a unique node that can be used in one or more RDF statements, but has no intrinsic name."

A classical use case, where blank nodes are applied, is the representation of complex

⁹<http://www.w3.org/TR/rdf-concepts/#section-blank-nodes>

data. It can be used to indirectly attach a resource to a consistent set of properties, which together represent a complex data, such as a postal address. The different fields of the complex data are represented as properties attached to the blank node. The ontology language OWL uses blank nodes to represent so-called *anonymous classes* such as unions or intersections of classes, or classes called *restrictions*, defined by a constraint on a property. And this is exactly the case in our Margherita pizza example. The toppings of the Margherita pizza are not defined in a <subject, predicate, object> way, such as <"Margherita", "hasTopping", "MozzarellaTopping"> (this way AquaLog could easily answer our question above), but via *restrictions on properties*. Let us have a closer look at the part of the ontology file, where the Margherita pizza is defined (RDF serialisation in XML format).

```

1  ...
2  <owl:Class rdf:about="#Margherita">
3    <rdfs:label xml:lang="pt">Margherita</rdfs:label>
4    <rdfs:subClassOf>
5      <owl:Restriction>
6        <owl:onProperty>
7          <owl:ObjectProperty rdf:about="#hasTopping"/>
8        </owl:onProperty>
9        <owl:allValuesFrom>
10         <owl:Class>
11           <owl:unionOf rdf:parseType="Collection">
12             <owl:Class rdf:about="#MozzarellaTopping"/>
13             <owl:Class rdf:about="#TomatoTopping"/>
14           </owl:unionOf>
15         </owl:Class>
16       </owl:allValuesFrom>
17     </owl:Restriction>
18   </rdfs:subClassOf>
19   <rdfs:subClassOf>
20     <owl:Restriction>
21       <owl:someValuesFrom>
22         <owl:Class rdf:about="#MozzarellaTopping"/>
23       </owl:someValuesFrom>
24       <owl:onProperty>
25         <owl:ObjectProperty rdf:about="#hasTopping"/>
26       </owl:onProperty>
27     </owl:Restriction>
28   </rdfs:subClassOf>
29   <rdfs:subClassOf>
30     <owl:Restriction>
31       <owl:onProperty>
32         <owl:ObjectProperty rdf:about="#hasTopping"/>
33       </owl:onProperty>
34       <owl:someValuesFrom rdf:resource="#TomatoTopping"/>
35     </owl:Restriction>
36   </rdfs:subClassOf>
37   <rdfs:subClassOf rdf:resource="#NamedPizza"/>
38   ...
39 </owl:Class>

```

Listing 5.1: RDF serialisation of the Margherita pizza description

As we can see from listing 5.1, the Margherita class is a subclass of several restrictions (*anonymous superclasses*, the superclasses illustrated in figure 5.7). These restrictions represent a complex data (namely the toppings definition) and are represented by a blank node, a resource (or node in an RDF graph), which is not identified by a URI.

We denoted before that AquaLog imposes a few requirements on the ontology in order to get an answer:

- The ontology must be structured as a directed labelled graph (taxonomy and relationships).
- The ontology must be populated (triples) in such a way that there is a short (two relations or less), direct path between the query terms when mapped to the ontology.

These assumptions conflict with the restrictions (owl:Restriction) depicted above, which are realised as blank nodes. In short: AquaLog does not understand restrictions modelled in OWL and can therefore not guess that the Margherita pizza has Tomato and Mozzarella as toppings.

This seems to be a handicap of this software system. Nevertheless, Lopez et al. [LUMP07] state that AquaLog was aimed to be built as a portable system targeted to the Semantic Web. Therefore, its developers chose to build an interface for querying ontology taxonomy, types, properties and instances, i.e. structures which are almost universal in Semantic Web ontologies. The assumptions AquaLog makes about the format of semantic information it handles leads to a *balance between portability and reasoning power*.

Thus, AquaLog *can not reason with ontology peculiarities*, such as the *restrictions* in the pizza ontology, and is therefore not fully qualified to meet our requirements.

Fortunately, there is a way to make AquaLog understand this very peculiarities, namely we can reason about these restrictions so that new statements in the form of <subject, predicate, object> can be inferred and added to the knowledge base. The tool which enables us to fulfil this task is the already mentioned *SwiftOWLIM*.

5.3.2 Enhancing reasoning capabilities

OWLIM is a high-performance semantic repository, implemented in Java and packaged as a Storage and Inference Layer (SAIL) for the Sesame RDF database (we will explain SAIL in section 5.3.2.2) [Onto07]. It is based on TRREE - a native rule entailment engine. The supported semantics can be configured through rule-set

definition and selection. Custom rule-sets allow tuning for optimal performance and expressivity. We will further discuss the adjustment of rule-sets in section 5.4.

OWLIM is available in two versions:

- *SwiftOWLIM*
Performs reasoning and query evaluation in-memory, while a reliable persistence strategy assures data preservation, consistency, and integrity. As stated in [Onto07], SwiftOWLIM is the fastest RDF(S) and OWL engine.
- *BigOWLIM*
Operates directly with binary persistence files, which allows it to scale to billions of statements. BigOWLIM is the only engine proven to support non-trivial OWL inference against 3 Billion triples [Onto07].

We use SwiftOWLIM, since it is an open-source library, published under the LGPL License¹⁰. It perfectly matches the needs we face in this work. BigOWLIM on the other hand is available under an RDBMS-like commercial licence on a per-server-CPU basis, and is neither free nor open-source.

In this section we already introduced the term *semantic repository*, which refers to a system for storage, querying, and management of structured data with respect to ontologies. Semantic repositories can be used as a replacement for the DBMS, offering easier integration of diverse data and more analytical power. In a nutshell, a semantic repository can dynamically interpret metadata schemata (e.g. RDF(S) files) *and* ontologies (such as the ones modelled in OWL), which define the structure and the semantics related to the data and the queries. Compared to the approach taken in the relational DBMS, this allows for easier changes to and combinations of data schemata, and for automated interpretation of the data.

5.3.2.1 TRREE Engine

TRREE¹¹ stands for *Triple Reasoning and Rule Entailment Engine*. It is implemented in Java and performs reasoning based on forward-chaining of entailment rules over RDF triple patterns with variables [Onto07]. TRREE's reasoning strategy is total materialisation. For further explanation of this strategy, please refer to Appendix C.

TRREE can be configured via so-called "rule-sets", which are basically sets of axiomatic triples and entailment rules determining the supported semantics. The version of TRREE used in SwiftOWLIM performs reasoning and query evaluation in-memory, which means that the full content of the repository is loaded and maintained in a proprietary representation in the main memory, allowing for rapid retrieval.

¹⁰The GNU Lesser General Public License, <http://www.gnu.org/copyleft/lesser.html>

¹¹<http://www.ontotext.com/trree/>

The TRREE rule compiler became part of the SwiftOWLIM Version 2.8.4 distribution (in this work we are using version 2.9.1), which allows the usage of custom rule-sets for inference. This way one can specify semantics which best fit the concrete application in terms of expressivity and performance. In the next chapter, we will further concentrate on the custom rule-sets.

As one can see in figure 5.1, a query sent to Sesame (via the Answer Engine) is forwarded to Sesame's SAIL API (which has a complete understanding of the storage model). The SAIL API then asks the OWLIM plug-in to forward the semantics of its request to the TRREE engine which, finally, executes the query and reasons about the knowledge base (or ontology).

Originally, AquaLog uses its own "Interpreter" which is interconnected between the Answer Engine and the Ontologies Knowledge Bases (see section 3 of [LUMP07]). But as we have shown, the expressivity and reasoning capabilities of this interpreter are very limited because of the close collaboration with the Sesame framework (through its Sesame plug-in) which only supports RDF(S) inferencing. Hence, we extended the reasoning capabilities by interconnecting the OWLIM plug-in, which supports reasoning for OWL files as well and allows for customisation of rule-set files to extend the expressivity of the reasoner.

5.3.2.2 The SAIL API

For persistent storage of RDF data, Sesame needs a scalable repository. At first, a Data Base Management System (DBMS) may come to mind, as these have been used for decades for storing large quantities of data. Since a variety of different DBMS have been developed over the last couple of decades, and it is impossible to know which way of storing the data is best fitted for which DBMS or which application domain, the Sesame architecture is built to keep itself DBMS-independent by concentrating all DBMS-specific code in a single architectural layer of Sesame: the *Storage and Inference Layer* (SAIL). Any particular SAIL implementation has a complete understanding of the storage model (e.g. the database schema in the case of an RDBMS). This SAIL is an application programming interface (API) that offers RDF-specific methods (for instance, methods for querying class and property subsumption, and domain and range restrictions) to its clients and translates these methods to calls to its specific DBMS.

But the data does not necessarily have to be stored in a DBMS. When setting up the Sesame server, some pre-configured SAILS are already installed and ready to use, e.g. SAILS for storing data in memory, or in files. SAIL implementations can also be stacked on top of each other, to provide functionality such as caching or concurrent access handling.

In our case, we are strongly interested in repositories which support inferencing. Hence, when setting up Sesame with its SAILS delivered by default, it would be the best to choose the MySQL SAIL which supports inferencing in contrast to native or in memory repositories.

However, as we already pointed out, the MySQL SAIL (used by AquaLog in its standard settings) has some inferencing restrictions which hinder us to get all the information we want from an *OWL file* (*Sesame supports inferencing for RDF Schema, not for OWL*). Remembering the introductory part, we mentioned that we have been delivered a parameter ontology by a prior work [Zalt08], and this parameter ontology is modelled using the OWL language. The solution to this problem is *SwiftOWLIM*, which is basically another SAIL and supports inferencing on OWL-files.

5.4 Custom Rule-Sets

As introduced in section 5.3.2, SwiftOWLIM is a high-performance semantic repository and based on the native rule entailment engine TRREE. Custom rule-sets (sets of axiomatic triples and entailment rules) allow tuning for optimal performance and expressivity and determine the supported semantics. It is now time to go into more detail and have a look at the rule language used by the TRREE engine, since we will create our own rules to enable AquaLog to handle restrictions.

A rule-set file basically consists of three major parts: Prefices, Axioms and Rules. These sections, which must appear in exactly this ordering, are each enclosed in curly braces { and }. Only the Rules section is mandatory.

- *Prefices*

The prefices of the common namespaces, such as the one for rdf, rdfs or owl, are defined here. Each pair of prefix/namespace should appear on a separate line. These defined prefices can later on be used in the Axioms and Rules section for qualifying resources with their local name only.

- *Axioms*

This section encloses a set of axiomatic triples to be asserted by default into the repository. They are usually used to describe the meta-level primitives which define the schema, such as rdf:type or rdfs:Class. Each line consists of exactly three node names, enclosed in < and > symbols and delimited by a white-space.

- *Rules*

This is the most interesting and important part of the rule-set file. A rule is defined via one or more *premises* and one or more *corollaries*, each defined via subject, predicate and object components. For these components, the user can either use a variable (single Latin letter that is not enclosed in < and >), a full URI or its short name (formed by a prefix as it was defined in the Prefices section, followed by ':' and its local name). Each triple component should be as well enclosed in < and > symbols (except the variables). Every rule obligatory starts with an ID which should appear in the beginning of a new line just after the *ID*: constant.

Each premise and corollary can be restricted via a *constraint*, stating that the value of one or more variables in the statement must not be equal to some specific full URI (or its short name) or to the value that is bound to another variable from the same rule. The left-hand side argument must be a variable, while the right-hand side value can be either a variable, a short name or a full URI. Constraints start with the *Constraint* constant, are delimited by comma and enclosed in [and] symbols. The premises and corollaries are delimited by a single line consisting of one or more - symbols.

The listing A.1 in Appendix A gives an impression on how such a rule-set file might look like.

A user who wants to utilise SwiftOWLIM does not have to create a complete rule-set file by his own, there are in fact some pre-defined rule-sets which come along with a respective distribution of SwiftOWLIM.

5.4.1 Semantics Supported by Default

The complexity, and thus the speed, of the inference can vary considerably across different rule-sets. The already mentioned pre-defined rule-sets are properly nested in each other regarding inference power. These rule-sets are identified by names and listed here in increasing expressivity order.

- *empty*
No sort of reasoning, i.e. OWLIM acts as a plain RDF store.
- *rdfs*
Support for the standard model theoretic RDFS semantics.
- *owl-horst*
OWL dialect close to OWL horst. For further explanation of OWL horst, refer to [Onto07].
- *owl-max*
A combination of most of the semantics of OWL Lite in combination with full compatibility with (support for) RDFS.

The most interesting and richest rule-set in this list is *owl-max*, which we will use as a starting point to create our own rule-set. Many OWL primitives used widely in ontologies on the Semantic Web are supported, for instance: *SymmetricProperty*, *TransitiveProperty*, *equivalentClass*, *sameAs*, *FunctionalProperty*, *allValuesFrom*, *someValuesFrom*, *unionOf*, *differentFrom*, *Thing* and many more.

OWLIM has an internal rule compiler that is used to configure TRREE with our custom set of inference rules and axioms. In order to get SwiftOWLIM to use our custom rule-set, we have to modify the mentioned owl-max rule-set - which is

basically a text file with the extension *.pie* - and specify the rule-set to use via the *ruleset configuration parameter* of the SAIL. The file will then be processed and its respective "inferences" will be generated and compiled.

The next chapter will clarify how exactly a SAIL in *Sesame* is to be configured and what *parameters* we have to set in order to get *AquaLog* working properly in the sense that it will use our custom rule-set and therefore be able to *understand restrictions* modelled in OWL.

"In theory there is no difference between theory and practice. In practice, there is."

- Bruce Schneier

6

Implementation

This chapter will describe all necessary steps which need to be done to get our architecture up and running. We will explain how to configure the aforementioned tools, what parameters we have to set, how the AquaLog system needs to be adjusted, and, most importantly, how exactly the custom rule-set file will look like, enabling us to query any ontology we want.

6.1 SwiftOWLIM SAIL Configuration

As discussed earlier, OWLIM is a specific plug-in (namely, a Storage And Inference Layer) for Sesame. To configure, run, and use OWLIM means to do so for a specific configuration of Sesame. There are numerous parameters which can be adjusted when using SwiftOWLIM. The table shown in Appendix B lists all these parameters together with a short description of their usage.

The parameters can be set via a very intuitive Sesame GUI, as shown in figure 6.1, and are finally stored in a configuration file named **system.conf**, which is located in the **WEB-INF** folder of the Sesame application. Remember that Sesame is running in a Tomcat servlet container and therefore the Sesame files are located in the **webapps** folder of Tomcat. For more information on how to install and configure SwiftOWLIM either in *embedded mode* (as a library, invoked in the same process as the application using it; this is the case here) or via *remote access* (running as a standalone server in a separate process, communication via RMI calls), please refer to section 7 of [Onto07].

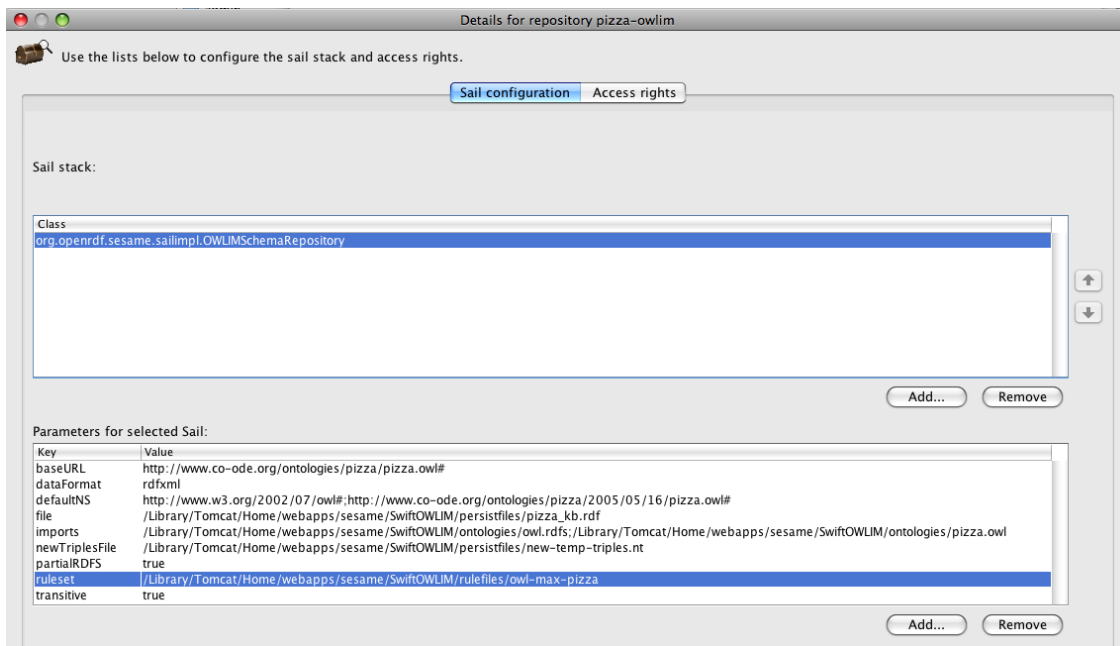


Figure 6.1: Configuration of the SAIL stack for the SwiftOWLIM SAIL

```

1 <repository id='pizza-owlim'>
2   <title>Pizza - OWLIM</title>
3   <sailstack>
4     <sail class='org.openrdf.sesame.sailimpl.
5       OWLIMSchemaRepository'>
6       <param name='imports' value='/Library/Tomcat/Home/
7         webapps/sesame/SwiftOWLIM/ontologies/owl.rdfs;/
8         Library/Tomcat/Home/webapps/sesame/SwiftOWLIM/
9         ontologies/pizza.owl' />
10      <param name='partialRDFS' value='true' />
11      <param name='dataFormat' value='rdxml' />
12      <param name='transitive' value='true' />
13      <param name='file' value='/Library/Tomcat/Home/webapps/
14        sesame/SwiftOWLIM/persistfiles/pizza_kb.rdf' />
15      <param name='ruleset' value='/Library/Tomcat/Home/
16        webapps/sesame/SwiftOWLIM/rulefiles/owl-max-pizza' />
17      <param name='baseURL' value='http://www.co-ode.org/
18        ontologies/pizza/pizza.owl#' />
19      <param name='newTriplesFile' value='/Library/Tomcat/
20        Home/webapps/sesame/SwiftOWLIM/persistfiles/new-temp-
21        triples.nt' />
22      <param name='defaultNS' value='http://www.w3.org
23        /2002/07/owl#;http://www.co-ode.org/ontologies/pizza
24        /2005/05/16/pizza.owl#' />
25    </sail>
26  </sailstack>
27  <acl worldReadable='true' worldWriteable='false' />
28 </repository>

```

Listing 6.1: Configuration of the SAIL stack for SwiftOWLIM

An example `<repository>` section that may appear in the `system.conf` configuration file of Sesame is depicted in listing 6.1 (reflects settings depicted in figure 6.1).

A further understanding of the used parameters is given by looking at the detailed description in Appendix B. However, two of those parameters should be considered in more detail since they affect the compilation of our custom rule-set and thus the compilation of the inferencers: `partialRDFS` and `transitive`.

6.2 Performance Optimisation Parameters

As [Onto07] states, there are several features in the specification of RDF(S) and OWL that cause the performance of a reasoning engine to decrease significantly because of "inefficient" entailment rules and axioms. Examples are as follows:

- The statement `<?X, rdf:type, rdfs:Resource>` is true for each URI node in any given RDF graph. Every URI node is a resource, and the engine should be able to entail and check this.
- All OWL classes are sub-classes of `owl:Thing`, thus the statement `<?X, rdf:type, owl:Thing>` applies for all individuals.
- The system should be able to infer that resources are classes and properties if they appear in schema-defining statements like `<?X, rdfs:subClassOf, ?Y>`.

Although these listed inferences are correct and important for the completeness of the formal semantics, they have a negative impact on the performance which may not be justified considering their utility in controlled environments. For instance, usually RDF schema and OWL ontologies are formalised so that their classes are explicitly introduced as instances of `rdfs:Class` or `owl:Class`. If this is the case, then the entailment of `<C, rdf:type, rdfs:Class>` each time when an instance of the class is defined by `<I, rdf:type, C>` is redundant.

The `partialRDFS`¹ parameter of OWLIM allows for switching on and off the use of rule-sets which are "optimised" in the sense that they do not contain the above mentioned inefficient axioms and rules.

The second parameter we have to mention is `transitive`.

If this parameter is set to `true`, the rule with the ID `owl_inv0f` is excluded.

TRREE's rule-compiler uses a single rule-set file to generate *four different inferencers*, corresponding to the combinations of the values of these two parameters. Note, however, that some RDF(S) resources are to be mentioned within an axiom or a rule for the TRREE engine to work properly. These are `rdf:type`, `rdfs:range`,

¹The name reflects earlier versions when there were only RDFS "optimisations". Since version 2.9.1 (which we use) of SwiftOWLIM similar optimisations are introduced also in the OWL support.

`rdfs:domain`, `rdfs:subClassOf`, `rdfs:Class` and `rdfs:subPropertyOf`.

The next section will explain how exactly we customise our rule-set and create our own rule to enrich the semantic support and make our system an even more sophisticated question answering machine.

6.3 Custom Rule-Set Creation

In section 5.4, we were introducing the basic assembly of a rule-set file, which comprises a `prefices-`, `axioms-`, and `rules-`part. Our new rule will be placed into this last part and compiled by the TRREE engine to infer new statements and add them to the knowledge base. This rule, which we call `owl_relationFromRestriction`, is presented in listing 6.4. But before we explain the idea of this rule, we first have to take a look at a rule which already exists in the *owl-max* rule-set file and by means of which we will clarify the difference between instances and classes. This is important to understand, otherwise one might run into problems when troubleshooting his own rule.

The rule we are talking about is called `owl_typeBySomeVal`, illustrated in listing 6.2.

```

1 Id: owl_typeBySomeVal
2 // Support for restrictions owl:onProperty of type
3 // owl:someValuesFrom. The support is limited to the
4 // inference of a restriction membership for nodes related
5 // to other nodes (values) of the corresponding class
6 // through the restricted property.
7
8   q  <rdf:type>          c
9   r  <owl:someValuesFrom> c
10  r  <owl:onProperty>    p
11  i  p  q
12  -----
13  i  <rdf:type>  r

```

Listing 6.2: Rule supporting restrictions `owl:onProperty` of type `owl:someValuesFrom`

This rule infers class membership² `i` to the restriction `r` for all nodes which are used as subjects of a statement (`<i, p, q>`) in which these subjects are linked to objects `q` - which are members of class `c` (the `someValuesFrom` argument) - via a property `p` (`onProperty` argument) such as the one used in the restriction `r`.

OWLIM does nothing to close the semantics in the opposite direction, e.g. it does not try to entail anything for the explicit members of the restriction. This means that it does not try to ensure or assert a fictive relation so that there exists such a relation between the instance which is member of the restriction and some instance of the class `c`.

²Note that `rdf:type` is used to define an instance of a certain class.

Having this in mind and looking at the Margherita definition in the pizza ontology, we now suggest the following:

First of all, we have to introduce a new property called `pizza:relatedTo`, which is to be used to 'propagate' the meaning of `hasTopping`, `hasSpiciness` and `hasBase` *from the realm of instances to the realm of classes*. This means that we have to add these statements:

- `<pizza:hasBase, pizza:relatedTo, pizza:useBase>`
- `<pizza:hasTopping, pizza:relatedTo, pizza:useTopping>`
- `<pizza:hasSpiciness, pizza:relatedTo, pizza:useSpiciness>`

The pizza ontology file should therefore be modified in the following way:

```

1  ...
2
3  <owl:ObjectProperty rdf:about="#hasBase">
4      <rdf:type rdf:resource="#owl:InverseFunctionalProperty"/>
5      <rdf:type rdf:resource="#owl:FunctionalProperty"/>
6      <rdfs:subPropertyOf rdf:resource="#hasIngredient"/>
7      <rdfs:range rdf:resource="#PizzaBase"/>
8      <rdfs:domain rdf:resource="#Pizza"/>
9      <owl:inverseOf rdf:resource="#isBaseOf"/>
10
11      <pizza:relatedTo rdf:resource="#useBase"/>
12
13  </owl:ObjectProperty>
14
15  <owl:ObjectProperty rdf:about="#hasSpiciness">
16      <rdf:type rdf:resource="#owl:FunctionalProperty"/>
17      <rdfs:range rdf:resource="#Spiciness"/>
18      <rdfs:comment xml:lang="en">A property created to be used with
19          the ValuePartition - Spiciness.</rdfs:comment>
20
21      <pizza:relatedTo rdf:resource="#useSpiciness"/>
22
23  </owl:ObjectProperty>
24
25  <owl:ObjectProperty rdf:about="#hasTopping">
26      <rdf:type rdf:resource="#owl:InverseFunctionalProperty"/>
27      <rdfs:comment xml:lang="en">Note that hasTopping is inverse
28          functional because isToppingOf is functional</rdfs:comment>
29      <rdfs:domain rdf:resource="#Pizza"/>
30      <rdfs:subPropertyOf rdf:resource="#hasIngredient"/>
31      <rdfs:range rdf:resource="#PizzaTopping"/>
32      <owl:inverseOf rdf:resource="#isToppingOf"/>
33
34      <pizza:relatedTo rdf:resource="#useTopping"/>
35
36  </owl:ObjectProperty>

```

```

35
36 ...
37
38 <!-- http://www.co-ode.org/ontologies/pizza/pizza.owl#relatedTo
   -->
39 <pizza:Property rdf:about="#relatedTo">
40   <rdfs:label xml:lang="pt">A property that relates one
      ObjectProperty to another (in pizza ontology)</rdfs:label>
41 </pizza:Property>
42
43 <!-- http://www.co-ode.org/ontologies/pizza/pizza.owl#useBase -->
44 <owl:ObjectProperty rdf:about="#useBase">
45 </owl:ObjectProperty>
46
47 <!-- http://www.co-ode.org/ontologies/pizza/pizza.owl#
   useIngredient -->
48 <owl:ObjectProperty rdf:about="#useIngredient">
49 </owl:ObjectProperty>
50
51 <!-- http://www.co-ode.org/ontologies/pizza/pizza.owl#useSpiciness
   -->
52 <owl:ObjectProperty rdf:about="#useSpiciness">
53 </owl:ObjectProperty>
54
55 <!-- http://www.co-ode.org/ontologies/pizza/pizza.owl#useTopping
   -->
56 <owl:ObjectProperty rdf:about="#useTopping">
57 </owl:ObjectProperty>
58
59 ...

```

Listing 6.3: Modification the the pizza ontology file

Secondly, we need to create a rule which *asserts the related relation* instead of the one used in the restriction definition (because that is actually used between instances, as we just explained). This rule is the one illustrated in listing 6.4.

```

1 Id: owl_relationFromRestriction
2   a <rdfs:subClassOf>      r [Constraint a != r]
3   r <rdf:type>             <owl:Restriction>
4   r <owl:onProperty>      p
5   p <pizza:relatedTo>     q
6   p <rdf:type>             <owl:ObjectProperty>
7   r <owl:someValuesFrom>  c
8 -----
9   a  q  c

```

Listing 6.4: Rule handling restrictions

It would be best for the reader to again take a look at listing 5.1 (the RDF serialisation of the Margherita pizza description) in order to properly follow and understand the assembly of this rule. The rule is to be interpreted in the following way:

We are looking for classes *a* which are subclasses of a restriction *r* (where *a* and *r* may not be equal³). This restriction restricts some class to a class *c* via an object property *p*, which is in turn related to a property *q*.

We can test our results right away using the Sesame Web-GUI and typing in the following SeRQL query⁴:

```

1 select pizza:Margherita, P, T
2 from {pizza:Margherita} P {T},
3     {Q} pizza:relatedTo {P}
4 using namespace
5     rdfs = <http://www.w3.org/2000/01/rdf-schema#>,
6     rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
7     pizza = <http://www.co-ode.org/ontologies/pizza/pizza.owl#>,
8     owl = <http://www.w3.org/2002/07/owl#>

```

Listing 6.5: SeRQL query to verify the success of the new rule

Figure 6.2 shows the query results before applying the new rule, figure 6.3 illustrates our results after applying it.

Logged in: **Test User** [log out]
Read actions: [SeRQL-S](#) [SeRQL-C](#) [RDQL](#) [RQL](#) [Extract](#) [Explore](#)
Repository: **Pizza - OWLIM** [select other]
Modify actions: [Add \(file\)](#) [Add \(www\)](#) [Add \(copy-paste\)](#) [Remove](#) [Clear](#)

Evaluate a SeRQL-select query


Your query:

```

select pizza:Margherita, P, T from
{pizza:Margherita} P {T},
{Q} pizza:relatedTo {P}
using namespace
rdfs = <http://www.w3.org/2000/01/rdf-schema#>,
rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>,
pizza = <http://www.co-ode.org/ontologies/pizza/pizza.owl#>,
owl = <http://www.w3.org/2002/07/owl#>

```

Response format: HTML Append namespaces Evaluate

 copyright © 2001-2006 Aduna BV

Query results:

<http://www.co-ode.org/ontologies/pizza/pizza.owl#Margherita>	P	T
---	---	---

0 results found in 6 ms.

Figure 6.2: SeRQL query results before application of OWLIM rule

As we can see in figure 6.3, new statements have been inferred from the restrictions, which have been added to the knowledgebase and would never have been asserted without the help of the rule created using SwiftOWLIM.

³In RDF, every class is sub-class of its own.

⁴Please refer to the online documentation available under <http://openrdf.org/doc/sesame/users/userguide.html#chapter-serql> to comprehend the syntax of SeRQL.

Logged in: **Test User** [\[log out\]](#)
Repository: **Pizza - OWLIM** [\[select other\]](#)
Read actions: [SeRQL-S](#) [SeRQL-C](#) [RDQL](#) [RQL](#) [Extract](#) [Explore](#)
Modify actions: [Add \(file\)](#) [Add \(www\)](#) [Add \(copy-paste\)](#) [Remove](#) [Clear](#)

Evaluate a SeRQL-select query

Your query:

select pizza:Margherita, P, T from
{pizza:Margherita} P {T},
{Q} pizza:relatedTo {P}
using namespace
rdfs = <http://www.w3.org/2000/01/rdf-schema#>
rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
pizza = <http://www.co-ode.org/ontologies/pizza/pizza.owl#>
owl = <http://www.w3.org/2002/07/owl#>

Clear

Response format:

HTML

Append namespaces

Evaluate

PDF

SVG

SH

XML


copyright © 2001-2006 [Aduna BV](#)

Query results:

	P	T
<http://www.co-ode.org/ontologies/pizza/pizza.owl#Margherita>	http://www.co-ode.org/ontologies/pizza/pizza.owl#useTopping	http://www.co-ode.org/ontologies/pizza/pizza.owl#CheeseTopping
http://www.co-ode.org/ontologies/pizza/pizza.owl#Margherita	http://www.co-ode.org/ontologies/pizza/pizza.owl#useTopping	http://www.co-ode.org/ontologies/pizza/pizza.owl#MozzarellaTopping
http://www.co-ode.org/ontologies/pizza/pizza.owl#Margherita	http://www.co-ode.org/ontologies/pizza/pizza.owl#useTopping	http://www.co-ode.org/ontologies/pizza/pizza.owl#TomatoTopping
http://www.co-ode.org/ontologies/pizza/pizza.owl#Margherita	http://www.co-ode.org/ontologies/pizza/pizza.owl#useBase	http://www.co-ode.org/ontologies/pizza/pizza.owl#PizzaBase


4 results found in 25 ms.

Figure 6.3: SeRQL query results after application of OWLIM rule



KIT
Karlsruhe Institute of Technology

70



The next step would be to test if the new inferred statements are recognised by AquaLog. Therefore, we ask the query engine one more time the same question, namely "what is topping of margherita". Surprisingly, we again did not get the expected answer in the form of a list of pizza toppings belonging to the Margherita pizza. Instead, we receive the same (error) message as last time. Having insured ourselves one more time that the new statements are added to the knowledgebase (see figure 6.3), we remember that the new statement uses the object property *useTopping* (and *useSpiciness*, *useBase*, respectively) instead of *hasTopping*. Thus, we adjust our ontology in the way that *useTopping* is in the list of *equivalent object properties* of *hasTopping*, making it an inverse property of *isTopping* as well.

Unfortunately, there seems to be a bug in the AquaLog system, causing it to fail at startup after having applied our changes by saving the ontology and hitting the "Send configuration to server button" in the SAIL configuration GUI of Sesame. The error message informed us about a `java.lang.OutOfMemoryError: Java heap space error`.

We should mention at this point, that there is not a final release of AquaLog declared as *stable*. Although it is open-source and the developers officially stopped the development process, we discovered numerous bugs during our usage. This might as well be another bug not yet discovered by the developers of AquaLog.

One of these bugs, for instance, concerns the ontology view delivered with the Web user interface of AquaLog. Depending on the kind of ontology used, the SeRQL-query that is responsible for the buildup of the hierarchical view of the ontology needs to be changed. The query used in AquaLog by default and our adaption is illustrated in the following listing.

```

1 // The original query - we call it Query 1
2 select distinct sc
3 from {sc} serql:directSubClassOf
4     {<http://www.w3.org/2000/01/rdf-schema#Resource>}
5 where not sc =<http://www.w3.org/2000/01/rdf-schema#Resource>
6     and isURI(sc)
7     and not namespace(sc) like "http://www.w3.org*"
8
9 //The query used for the pizza ontology - we call it Query 2
10 select distinct sc
11 from {sc} serql:directSubClassOf
12     {<http://www.w3.org/2002/07/owl#Thing>}
13 where not sc =<http://www.w3.org/2002/07/owl#Thing>
14     and isURI(sc)

```

Listing 6.6: SeRQL query to find out the root classes of an ontology

The major difference is, that when using an OWL ontology, we obviously should query for *Thing*, since all OWL classes are sub-classes of *owl:Thing*, and not *Resource*.

When using an OWLIM SAIL, we also have to consider the setting of the parameter

`partialRDFS`. We elaborated on this parameter earlier in section 6.2. A summary of our results is given in the following list:

- *Query 1 used with `partialRDFS` set to true*
The only class which is listed is the class **Property**.
- *Query 1 used with `partialRDFS` set to false*
The only class which is listed is the class **Nothing**.
- *Query 2 used with `partialRDFS` set to true*
The correct hierarchical view is listed, as illustrated in figure 6.4.
- *Query 2 used with `partialRDFS` set to false*
The only class which is listed is the class **Nothing**.



Figure 6.4: Hierarchical view of the pizza ontology in AquaLog

Note that during initialisation of AquaLog, the process of creating this hierarchical view takes place. After the view has been created, it is saved as a text file in the `descriptions` folder of the AquaLog application (which resides in the Tomcat `webapps` folder), named after the respective ontology name. If this file already exists, AquaLog reads this file and does not create it again in order to save start-up time.

There is one more peculiarity of OWL files we have to mention and which deals with a correct depiction of the ontology hierarchy in the AquaLog ontology viewer. This peculiarity has to do with the OWL construct `equivalentClass`, which we will discuss in the next section.

6.4 Problems with SeRQL-Directives

The Java class contained in the AquaLog system, which serves as an interface for the Sesame framework, is called `SesamePlugin.java`. All methods and functions which deal with data read from or written to a Sesame repository are placed in here. The MySQL SAIL, which is one of the few SAILS delivered with a SwiftOWLIM distribution that is capable of RDF(S) inferencing (and which we first used to ask questions against the pizza ontology), is not able to find classes modelled as `equivalentClass`, but only if these equivalent classes are not explicitly modelled as a

sub-class of another class (which is most of the time the case in the pizza ontology). This means, if there is a class modelled as an equivalent class *and also* has a "`<rdfs:subClassOf ...>`" statement, then this SAIL can find it, otherwise not. For instance, the following query was not able to find the classes `MeatyPizza`, `InterestingPizza`, `CheeseyPizza` and so on, which are modelled as sub-classes of class `Pizza`.

```

1  select distinct p
2  from {p} rdfs:subClassOf {<http://www.co-ode.org/ontologies/pizza
   /2005/05/16/pizza.owl#Pizza>}
3  where isURI(p)

```

Listing 6.7: SeRQL query to find sub-classes of class `Pizza`

This seemed odd at first, then we figured, that the MySQL SAIL is only a RDF(S) inferencer, but the equivalency is an OWL language construct. Hence, this SAIL can not understand this construct and ignores it (as well as the class itself).

The SwiftOWLIM SAIL, on the other hand, is able to do OWL inferencing and thus recognises `MeatyPizza`, `InterestingPizza`, etc. as sub-classes of class `Pizza`. BUT: The SwiftOWLIM SAIL can not recognise these equivalent classes as *direct sub-classes* of class `Pizza`, because then premise 1) and 2) of the definition of `serql:directSubClassOf` are violated. There are three so-called *directives* in SeRQL. Please refer to the online documentation of SeRQL⁵ to follow this argumentation.

`X serql:directSubClassOf Y`. This relation holds for every `X` and `Y` where:

- 1) `X rdfs:subClassOf Y`
- 2) `X != Y`
- 3) There is no class `Z` (`Z != Y` and `Z != X`) such that `X rdfs:subClassOf Z` and `Z rdfs:subClassOf Y`

Thus, equivalent classes can not be found because they are always sub-class of their own *and their* equivalent classes ($\rightarrow X = Y$).

For that reason, we have been thinking about a suitable and working query which substitutes the `serql:directSubClassOf` directive. Then we changed the Java code at the according location in `SesamePlugin.java` (the method is called `GetDirectSubClasses()`), as illustrated in listing 6.8.

```

1  ...
2
3  /*serql="select distinct c,l " +
4      "from {c} serql:directSubClassOf {<"+concept+">}, [{c}
       rdfs:label {l}] " +
5      "where isURI(c) and not c =<"+concept+">";*/
6
7  serql="select distinct c " +

```

⁵<http://openrdf.org/doc/sesame/users/userguide.html#chapter-serql>

```

8      "from {c} rdfs:subClassOf {<" + concept + ">} " +
9      "where isURI(c) and not c=<" + concept + "> " +
10     "minus " +
11     "select distinct c " +
12     "from {c} rdfs:subClassOf {c2} rdfs:subClassOf {<" + concept
      + ">} " +
13     "where isURI(c) and
14           isURI(c2) and
15           not c2=<" + concept + "> and
16           not c=c2";
17
18 (... the rest stays the same)

```

Listing 6.8: SeRQL query to find the direct sub-classes of a certain class

With that SeRQL-query we managed to get a correct hierarchical class view in the AquaLog ontology viewer.

Another SeRQL directive for which we had to find a substitution is `serql:directType`. AquaLog uses this directive in its `getInstances()` method in `SesamePlugin.java`. The change is illustrated in listing 6.9.

```

1  ...
2
3  /* serql="select distinct i,l " +
4      "from {i} serql:directType {<"+classURI+">}, [{i}
5          rdfs:label {l}] " +
6          "where isURI(i)"; */
7
8  serql="select i,l " +
9      "from {i} rdf:type {j} rdfs:subClassOf {<" + classURI + "
10         >}, [{i} rdfs:label {l}] " +
11         "where isURI(i) " +
12         "minus " +
13         "select i,j,l from {i} rdf:type {j} rdfs:subClassOf {<" +
14             classURI + ">}, [{i} rdfs:label {l}] " +
15             "where isURI(i) and j!<" + classURI + ">";
16
17 (... the rest stays the same)

```

Listing 6.9: SeRQL query to find the instances of a certain class

The reason why the `serql:directType` directive is not working is again the OWL language construct `<owl:equivalentClass>` (rule 2 is violated). With this new SeRQL-query, all instances for each and every class can be found and AquaLog is fortunately able to answer the query "show me countries", returning a list of all countries modelled as instances of the class `Country`.

Recapitulatory, we can say that we were able to show that we can extend the expressiveness supported by Sesame (and, thus, AquaLog as well) with SwiftOWLIM, to be more precise, with the rule-sets offered by SwiftOWLIM, and minor changes

to the AquaLog code. Any modelling aspect and peculiarity of an ontology not understood by Sesame because of its restriction to RDF(S) inferencing can be reasoned about and hence, new statements can be asserted and added to the knowledgebase, making it Sesame easier to handle this information. AquaLog is in principle able to 'see' everything Sesame 'sees'. But due to several implementation bugs still existing in the AquaLog system, it is not able to display this information right away. Assumptions made by AquaLog about the design of the underlying ontology are too strict to handle any peculiarity, hence, reasoning power (offered by AquaLog from scratch) is sacrificed to portability, as stated by Lopez et al. [LUMP07]. However, as it turns out, the aimed portability is not reached by far as it takes a substantial amount of time to figure out how to adjust AquaLog to 'understand' an ontology which is not modelled in the way its developers sought it to be.

However, there is an ontology which might meet the assumptions made by AquaLog on the ontology design. The results of this diploma thesis are supposed to be utilised in TEXO⁶, a BMWi⁷ funded project aiming at research in business webs in the so-called Internet of Services. In the course of this project, a service ontology is being developed, trying to capture amongst others service descriptions for so-called EcoCalculator services, MaterialLookup services, and Monitoring services. In the following section, we will test our question answering system against this ontology.

6.5 Applicability of the TEXO Ontology

Figure 6.5 illustrates the status quo of the TEXO ontology as visualised by the tool Protégé. The ontology is not very dense populated at the time of writing this thesis because it is still under development. But we should be able to work with this ontology since it is properly modelled in a way so that the assumptions made by AquaLog over the design of ontologies are met.

We can see from the above figure that there are several members of the class `EcoCalculatorServiceDescription` (which is a sub-class of `ServiceDescription`), namely `eco-calculator`, `eco-it`, `ecochexx`, and `oekorechner`. A closer look to the member `eco-calculator` reveals that it is `offeredBy` `cde_provider`, `obeys` `eco-label-decree`, is `composedOf` `M`, `I`, and `MS`, and `obeys` `aus-eco-label-2007` (a fictive Australian eco certificate).

It would be interesting to see how well AquaLog can handle this ontology by asking some meaningful questions. At first, we are pleased to see that the ontology viewer of AquaLog correctly illustrates the ontological hierarchy. Then we ask the following question: "show me eco calculator service descriptions", expecting the list of members we can see in figure 6.5. Unfortunately, we receive the message "There are not instances or classes found in the ontology" from the system.

⁶<http://theseus-programm.de/scenarios/de/texo>

⁷<http://www.bmwi.de>

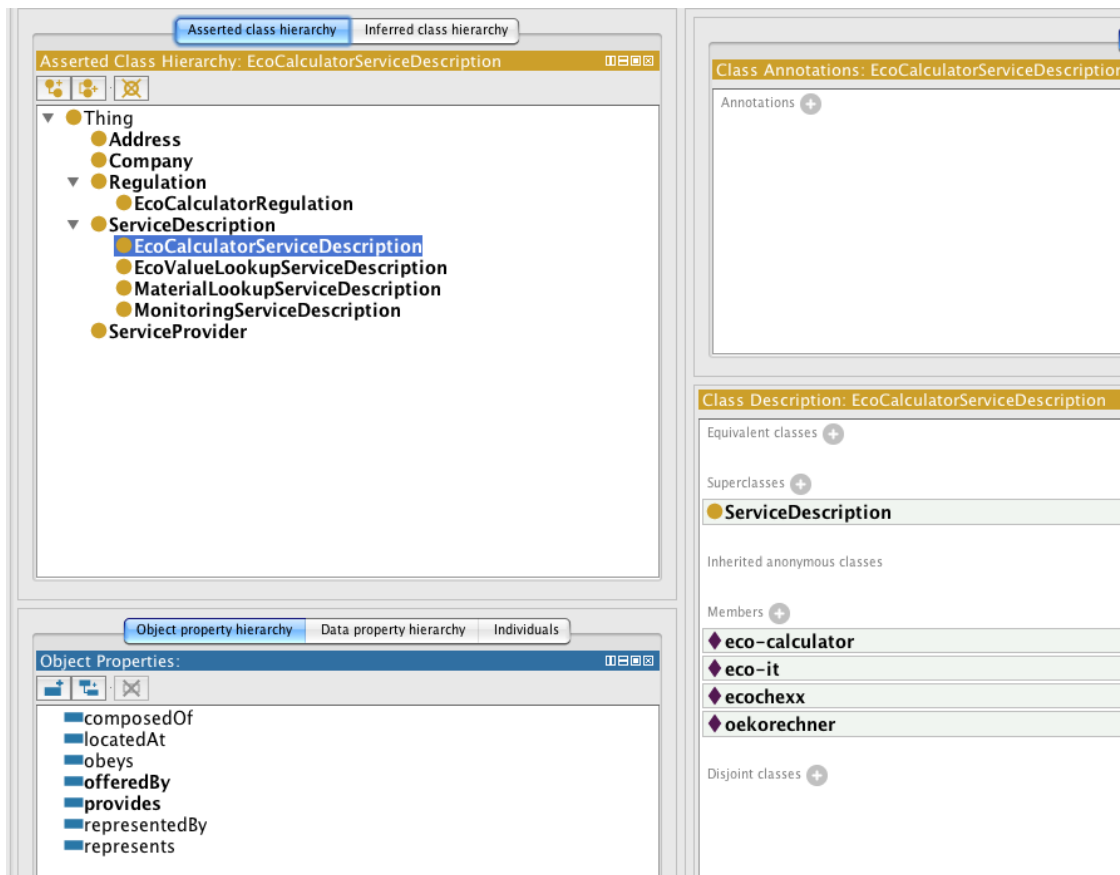


Figure 6.5: The TEXO ontology as illustrated by Protégé

Omitting the last word "descriptions", we then ask "show me eco calculator services", resulting in the ontology triple $\langle \text{who/what is, eco-calculator, [DESCRIPTION]} \rangle$ and the information about the member `eco-calculator`, as depicted in figure 6.6.

The information found about:

eco-calculator.:	
Instance	eco-calculator
Home ontology	texo-owlim
Instance of	EcoCalculatorServiceDescription
type	ServiceDescription , EcoCalculatorServiceDescription , Resource , Thing
comment	TollerService.
label	EcoCalculator,EcoCalculator
sameAs	eco-calculator , eco-calculator
obeys	aus-eco-label-2007 , aus-eco-label-2007 , eco-label-decree
offeredBy	cde_provider
composedOf	MS , MS , I , I , M , M
labels	Eco Calculator

Figure 6.6: Information about the eco-calculator presented by AquaLog

Although this is not exactly what we asked for (we expected a list of all the members of an eco calculator description), the system acted correctly since it identified the term "eco calculator services" as the ontology concept `eco-calculator`, which is self-evident, and presents all information found about this concept.

The next question which came into our mind was "who offers eco calculator service". This time, we get an error message from the Relation Similarity Service, stating: "There was an error while trying to match the instances (Name of the class person in the configuration file for queries type -who- is not found in the ontology)." This message brings us to the idea of customising the configuration file `query_properties.xml` (we will take a closer look at the configuration files in section 6.6). Here we declare that the question term "who" shall correspond to "ServiceProvider" instead of "Person". After restarting AquaLog, we asked the same question once more and sadly failed again to get a satisfying answer, as we can see from figures 6.7 and 6.8.

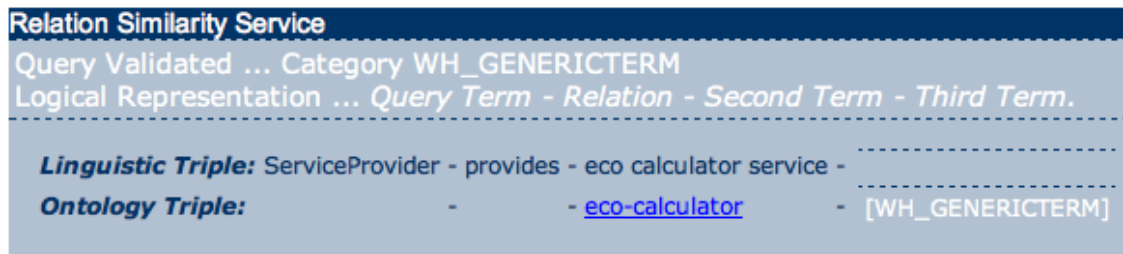


Figure 6.7: Answer drawn from the TEXO ontology

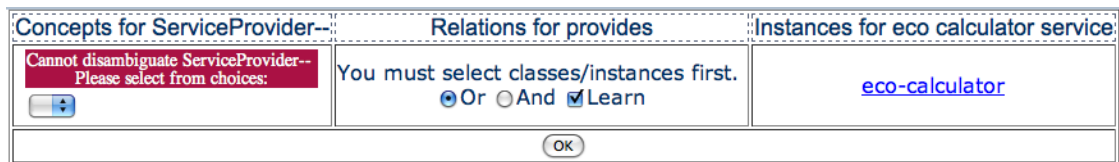


Figure 6.8: Answer drawn from the TEXO ontology

Although the system was able to map the question term "who" correctly to "ServiceProvider", it was - for a reason we are not aware of - not able to return an answer to a really obvious question. We were expecting to receive the answer, that `cde_provider` (which is an instance of `ServiceProvider`) provides the `eco-calculator`, because this is explicitly modelled in the ontology as illustrated in figure 6.9.

As we can see from the various examples given in this and the previous sections, there are still some problems with the AquaLog implementation, some of which are obvious and can be sailed around, some of which are not comprehensible at the moment. Nevertheless, we have demonstrated the great potential of our question answering system, which is a combination of the AquaLog system and our extension formed by the SwiftOWLIM plug-in and its modified rule-set file, respectively.

We will now have a closer look at the configuration files, since we already introduced the use of the `query_properties.xml` file in this section.

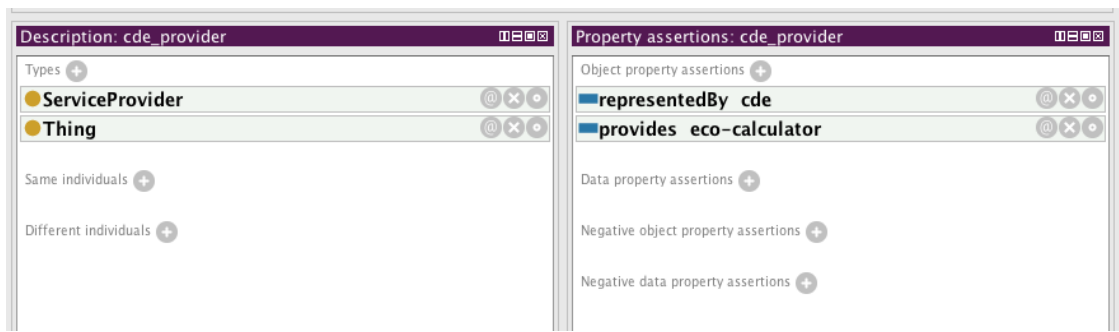


Figure 6.9: Information about the instance `cde_provider`

6.6 Customising Configuration Files

We have mentioned several times that AquaLog is a portable software system. AquaLog allows for its customisation by editing configuration files in order to adapt it to a new domain. The configuration parameters in these files are required to initialise the AquaLog server. The two most important files are listed below:

- *service_properties.xml*
Ontology name and server, login details if necessary and the name of the plugin can be configured here. Optionally, the main concepts of interest in an ontology can be specified.
- *query_properties.xml*
This file holds the ontology-dependent parameters which specify that the term "who", "where", "when" corresponds, for example, to the ontology terms "person/organisation", "location" and "time-position", respectively. Remember section 5.1.2, where we presented the exemplary question "who is the secretary in Knowledge Media Institute?" and showed that RSS was able to figure out that *who* could be pointing to a "person". Those question indicators are used in the set of annotations returned by GATE which was extended by the AquaLog development team.
- *database_properties.xml*
There is a database, called "learningmechanismdb", to automatically store the learned user jargon in a lexicon through the learning mechanism. MySQL must be installed. A script called "script_create_LM_tables.sql" needs to be run in order to create the database and tables to be used by the learning mechanism, which automatically updates the tables "lexicon" and "relations" (the last one links the user to the lexicon vocabulary). The user log-in information is stored in the table "users". If desired, one can manually update the table "termlexicon" to add new vocabulary. The location of the database is specified in database_properties.xml (the default setting is: localhost=3306, user=root with no password).

The independence of the application from the ontology is guaranteed through the first two configuration files.

There are other files which need adjustment in order to run the system on a certain machine (such as files stating the path to the AquaLog installation directory, absolute path to the WordNet library, proxy settings, etc.), but those are system-dependent parameters which we will not discuss. With every distribution of AquaLog, there comes a "Installation steps for AquaLog2 server.doc" document, guiding the user through the setup process to get AquaLog up and running.

An example how a service_properties.xml and a query_properties.xml file might look like is given in listings 6.10 and 6.11, respectively.

```

1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <CONFIGURATION>
4    <REPOSITORY>
5      <ONTOLOGY>pizza-owlim</ONTOLOGY>
6      <SERVER>http://localhost:8080/sesame</SERVER>
7      <PORT>8080</PORT>
8      <PLUGIN_MANAGER>WEB-INF/aquaplugins/</PLUGIN_MANAGER>
9      <ONTOLOGY_PLUGIN>sesame</ONTOLOGY_PLUGIN>
10     <LOGIN>testuser</LOGIN>
11     <PASSWORD>sesame</PASSWORD>
12     <CONCEPTS></CONCEPTS>
13     <PRETTY_NAMES></PRETTY_NAMES>
14     <TYPE>OWL</TYPE>
15   </REPOSITORY>
16 </CONFIGURATION>

```

Listing 6.10: Example of a service_properties.xml file

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <CONFIGURATION>
3    <WHO>person,organization</WHO>
4    <WHERE>location</WHERE>
5    <WHEN>time-position</WHEN>
6  </CONFIGURATION>

```

Listing 6.11: Example of a query_properties.xml file

Up to now, the pizza ontology and the TEXO ontology have been the only ontologies we tested our question answering system against. But as we stated in the beginning of our thesis, we actually want to test our system with a completely different ontology, namely the one parameter ontology which served as a starting point for our work. The applicability of this ontology to our question answering system will therefore be discussed in the following section.

6.7 Applicability of the Parameter Ontology

In the beginning we introduced the overall goal of this work, which is to facilitate the search for Web Services provided by SAP in its Enterprise Service Repository with the help of semantic technologies. We explained that the descriptions of these Web Services have been analysed in a prior work [Zalt08] to generate a service ontology for the SAP domain. With the help of resources such as the SAP-own glossary SAPterm and Business Object descriptions, a parameter ontology was created which served as a starting point for our work.

After we extensively tested our question answering system with the pizza ontology and examined the details of the part of the system belonging to AquaLog, we had to unfortunately realise, that this parameter ontology will not work with our system at all. The reason for this is the simple fact that not a single object property is included in this ontology.

The ontology was automatically created with the help of the WSDL files of the respective Web Services, describing what each and every Web Service is doing, the Business Object descriptions, explaining what a certain Business Object is all about, and the SAP-own glossary SAPterm, explaining every possible term related to the SAP world. Out of these resources, a hierarchical parameter ontology has been created, which indeed displays the hierarchy of SAPterms used in these Web Services, but without relating these terms together via object properties.

Thus, our system will not be able to answer any meaningful question asked against this ontology. Therefore, we assert that the parameter ontology has to be completely redesigned for our question answering system to be able to generate appropriate query triples and ontology triples and get reasonable answers in return. Generally, we state that an ontology modelled without any object property does not make sense at all and is useless for any question answering system allowing natural language queries.

If we would have to categorise ontologies on the basis of their design, then we would propose three different categories:

- Ontologies which are strictly modelled in the three-terms way <subject, predicate, object>. For every class modelled in the ontology, there is another class related to it via an object property (the predicate).
- Ontologies using restrictions, such as the pizza or wine ontology. These kind of ontologies use restrictions in the form of anonymous superclasses.
- Ontologies which do not use object properties at all.

We want to emphasise that this list is not an official categorisation proposed by any committee, but our own experience during this thesis. This brings us back to section 3.11, where we already discussed different quality measures of ontology design.

Summary and Conclusions

In this thesis we presented an approach to search for semantically enhanced data with the help a question answering tool named AquaLog and a semantic repository named SwiftOWLIM. The combination of these tools results in a software system which seems to be very powerful, but has its limitations as well.

On the one hand, it is able to accept questions formulated in natural language and make sense out of these query terms by applying natural language processing resources and trying to map query terms to concepts in an underlying ontology. Furthermore, the AquaLog system claims to be portable in the sense that it allows the user to choose an ontology and then ask queries with respect to the universe of discourse covered by the ontology, while keeping configuration time required to customise the system for a particular ontology to a minimum.

On the other hand, it makes assumptions on the design of ontologies which hinder the question answering machine to deliver meaningful results when querying ontologies with certain peculiarities not expected by AquaLog. The assumptions AquaLog makes about the format of semantic information it handles is supposed to lead to a balance between portability and reasoning power. However, we think that this system is neither portable enough nor does it support enough reasoning power to handle well-known ontologies such as the wide-spread pizza or wine ontology.

For this reason, we were doing further research to discover a solution which enhances the reasoning power of AquaLog, while being easy to integrate in the existing system. We then discovered SwiftOWLIM, a so-called semantic repository, which serves as a plug-in for the Sesame framework. Since AquaLog uses a Sesame plug-in itself and is relying on the expressivity of Sesame, we could enhance the overall expressivity of our system by customising a so-called rule-set offered by SwiftOWLIM. With the help of

rules we are able to reason about almost any peculiarity of an ontology and therefore assert new statements which are then added to the knowledgebase. This again results in more meaningful results to questions formulated in natural language, thus, enhancing our system to an even more sophisticated question answering machine.

We adducted several ontologies and tested their applicability to our question answering machine, resulting in a discussion about the advantages and drawbacks of our system concerning its ability to deliver meaningful results to simple questions formulated in natural language.

8

Outlook

Throughout this work, we have mentioned starting points to further develop and enhance our established solution of a sophisticated question answering machine.

First of all, we mentioned the JAPE grammars in section 5.1.1. JAPE is an expressive, regular expression based rule language offered by GATE. The Linguistic Component of AquaLog relies on these grammars to recognise e.g. question terms or intermediate representations of the query. One could extend the Linguistic Component to increase AquaLog's linguistic abilities by dealing with and creating new JAPE grammar rules. This has a high potential to further enhance the expressivity of our question answering system.

Secondly, the rule-set file is the central location to sail around the restrictions imposed by the AquaLog system. Any peculiarity of an ontology, which AquaLog may not be able to handle, can be made 'understandable', resulting in a higher acceptance of this system when meaningful results to queries are delivered, which otherwise could not be delivered.

Thirdly, the Sesame plug-in of AquaLog relies on the Sesame version 1.2.7. From version 2.0 on, Sesame supports queries formulated in SPARQL-syntax. SPARQL is a SQL-like query language similar to SeRQL, but more expressive. In order to adapt our question answering system to Sesame 2.0, the SesamePlugin.java file would have to be adapted to the new, completely redesigned API used by Sesame 2.0.

Last but not least, we have mentioned that AquaLog still seems to have some bugs which causes the system not to work the way even an experienced user would expect it to. During our work, the main architect of AquaLog, Vanessa Lopez, was more than helpful when trying to understand the working mechanisms of this system. We ourselves have discovered several bugs which have been fixed by Vanessa. She would for sure be glad to get more feedback on her program and, thus, all the problems

and bugs we were not able to solve due to the limited amount of time to write this thesis could be solved by consulting her via e-mail, for instance.

In chapter 4, we were introducing another question answering tool named *ORAKEL*. It would be interesting to analyse this tool as well and compare its reasoning capabilities and user friendliness to AquaLog. We could imagine a user study in which a comparison between these tools is covered.

Furthermore, it would be interesting to see how well our question answering system really works, if all bugs, which are still residing in the AquaLog system, are fixed and the TEXO ontology is well developed. Again, a user study would be of big interest, showing on the one side the real potential of our question answering system, and on the other side the level of user acceptance.

A

Appendix A

Listing A.1 gives an impression on how a basic rule-set used by the TRREE engine could look like.

```
1  Prefices
2  {
3      rdf : http://www.w3.org/1999/02/22-rdf-syntax-ns#
4      rdfs : http://www.w3.org/2000/01/rdf-schema#
5      owl : http://www.w3.org/2002/07/owl#
6      xsd : http://www.w3.org/2001/XMLSchema#
7  }
8
9  Axioms
10 {
11 // RDF axiomatic triples
12     <rdf:type> <rdf:type> <rdf:Property>
13     <rdf:subject> <rdf:type> <rdf:Property>
14     <rdf:predicate> <rdf:type> <rdf:Property>
15     <rdf:first> <rdf:type> <rdf:Property>
16     <rdf:rest> <rdf:type> <rdf:Property>
17     <rdf:nil> <rdf:type> <rdf:List>
18
19 // RDFS axiomatic triples
20     <rdf:type> <rdfs:domain> <rdfs:Resource>
21     <rdfs:domain> <rdfs:domain> <rdf:Property>
22     <rdfs:range> <rdfs:domain> <rdf:Property>
23     <rdfs:subPropertyOf> <rdfs:domain> <rdf:Property>
24     <rdfs:subClassOf> <rdfs:domain> <rdfs:Class>
25     <rdfs:label> <rdfs:domain> <rdfs:Resource>
26
27 // OWL trivial statements
```

```

28     <owl:sameAs>      <rdf:type> <owl:TransitiveProperty>
29     <owl:sameAs>      <rdf:type> <owl:SymmetricProperty>
30     <owl:inverseOf> <rdf:type> <owl:SymmetricProperty>
31 }
32
33 Rules
34 {
35 // This rule (Rule_Id) just serves as a template on
36 // how a rule would look like
37     ID: Rule_Id
38     < Premise #1 >
39     < Premise #2 >
40     ...
41     < Premise #n >
42     -----
43     < Corollary #1 >
44     < Corollary #2 >
45     ...
46     < Corollary #n >
47
48 // A concrete rule
49     ID: rdfs_subMissesPropDomain
50     p <rdfs:subPropertyOf> q           [Constraint p != q]
51     q <rdfs:domain> c
52     -----
53     p <rdfs:domain> c           [Constraint p != q]
54
55 // Another concrete rule
56     Id: rdfs9
57     a <rdf:type> x
58     x <rdfs:subClassOf> y           [Constraint x != y]
59     -----
60     a <rdf:type> y
61 }

```

Listing A.1: An example of a rule-set file

B

Appendix B

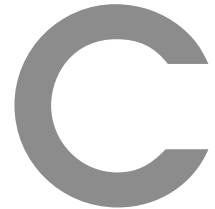
The OWLIM SAIL configuration parameters, with their default and allowed values, together with a short description of each parameter [Onto07].

Name	Default	Allowed Values
file	No default value	Any valid file name
Specifies the NTriples (.NT) file, where the repository contents are persisted (see [Onto07], section 3 for the persistency strategy and the role of this file). Example: <param name="file" value="./kb/kb.nt"/>		
dataFormat	ntriples	rdxml, turtle, ntriples
Specifies the serialisation format for the main persist file. Example: <param name="dataFormat" value="ntriples"/>		
compressFile	No	yes, no
Specifies whether a compression on the file will be used. Example: <param name="compressFile" value="no"/>		
noPersist	false	false, true

Name	Default	Allowed Values
ruleset	owl-horst	owl-max, owl-horst, rdfs, empty
<p>Specifies the set of axioms and entailment rules used for inference, which determines the supported semantics. OWLIM is packaged with four preconfigured sets, whose names are valid values of this parameter. (See section 5.4.1 for further information)</p> <p>Example:</p> <pre><param name="ruleset" value="owl-max"/></pre> <p>Note: If the value of this parameter does not match any of the allowed values, then it is considered to be a filename of a custom rule-set. The '.pie' extension is appended in case if not present. Then, if such a file exists it is processed, compiled and loaded dynamically so the TRREE engine can perform inference based on the rules defined there.</p>		
partialRDFS	true	false, true
<p>This parameter switches on (when set to <i>true</i>) and off few performance "optimisations" of the RDFS and OWL inference, as described in section 6.2. Its purpose is to suspend part of the entailments, which are useless for many datasets and applications, but require considerable reasoning resources. This optimisation has no effect when the ruleset parameter is set to empty.</p>		
indexSize	4.000.000	Limited by the memory available to the Java virtual machine. Values lower than 100.000 are ignored and the default one is used instead.
<p>Controls the initial size of the primary triple index. Its default value is convenient for repositories that hold approx. 5.000.000 explicit statements. One should alter this value in case that the target size differs considerably. A smaller value of indexSize will reduce the amount of memory used for the index, but such setting will slow down the repository operation as the volume of the data grows up. The amount of memory (in bytes) used for this index can be calculated as $20 \times \text{indexSize}$. With the default setting OWLIM allocates 80MB of memory for its primary triple index.</p> <p>Example:</p> <pre><param name="indexSize" value="4000000"/></pre>		
newTriplesFile	No default value	Any valid file name

Name	Default	Allowed Values
	<p>The parameter specifies the name of a file where OWLIM temporarily stores the new triples recently added to the repository. It is used in case of abnormal termination, so that during the initialisation its contents will be automatically added to the repository right after the contents of the main persist file (see parameter <code>file</code>). In case that OWLIM was properly shut down or successfully synchronised with the main <code>persist</code> file, the contents of the <code>new-triples-file</code> becomes superfluous, since each triple, mentioned there, is already included in the main <code>persist</code> file. If this parameter is omitted in the configuration of SAIL, the backup strategy is set to <code>off</code> and the repository and the repository contents will persist only in case the SAIL was properly shutdown. This would happen if the repository <code>shutdown()</code> method is invoked.</p> <p>Example:</p> <pre><param name="new-triples-file" value="./kb/new-triples.nt"/></pre>	
<code>baseURL</code>	No default value	Any valid URL
	<p>Specifies the default namespace for the main persist file. Non-empty namespaces are recommended, because their use guarantees the uniqueness of the anonymous nodes that may appear within the repository.</p> <p>Example:</p> <pre><param name="baseURL" value="http://www.ontotext.com/kim/2004/12/wkb#"/></pre>	
<code>imports</code>	No default value	Semicolon-delimited list of file names
	<p>A list of schema files which will be imported - all the statements found in these files will be loaded in the repository and will be treated as read-only. The serialisation format is assumed to be RDFS/XML, unless the file has a <code>.NT</code> extension.</p> <p>Example:</p> <pre><param name="imports" value="owl.rdfs;protons.owl;protont.owl"/></pre>	
<code>defaultNS</code>	None	Semicolon-delimited list of URLs. The number and order should match this in the <code>imports</code> parameter.
	<p>Specifies the default namespaces for each of the imported files.</p> <p>Example:</p> <pre><param name="defaultNS" value="[namespaces list for the imported files]"</pre>	

Name	Default	Allowed Values
transitive	false	false, true
<p>Switches 'on' an experimental, backward-chaining based implementation for transitive, inverse and symmetric properties or combination of those. In this mode OWLIM is relatively slow, but allows for handling of data which contain very long chains of resources connected with transitive (or/and symmetric) properties or inverse of such, thus avoiding materialisation of $O(N^2)$ statements that represent the whole transitive closure of such properties.</p> <p>Note: The "transitive" mode of TRREE is not thread safe, so the basic multi-threading code introduced in this version of OWLIM does not support it.</p>		



Appendix C

There are two principle strategies for rule-based inference, namely:

- *Forward-chaining*
Starting from the known facts (the explicit statements), inference is performed in an inductive way. The goal is either to compute the *inferred closure*¹, or to infer a particular sort of knowledge (e.g. the class taxonomy) as well as to answer a particular query.
- *Backward-chaining*
Starting from a particular fact or a query, it is geared towards its verification as well as getting all possible results, using deductive reasoning. In a nutshell, the reasoner decomposes (or transforms) the query (or the fact) into simpler (or alternative) facts, which are available in the KB or can be proven through further recursive transformations.

So, why are we mentioning these inference strategies? Well, a repository which performs total forward-chaining tries to make sure that, after each update to the KB, the inferred closure is computed and made available for query evaluation or retrieval. This strategy is generally known as *materialisation*. An inference strategy which follows this paradigm and *also* takes into account monotonic entailment² is

¹*Inferred closure* is defined as follows: A given KB (or a graph of RDF triples) is extended with all the implicit facts (triples), which could be inferred from it, based on the enforced semantics.

²In case of monotonic logic, adding new explicit facts (statements) to the KB (repository) can cause new implicit facts to extend its inferred closure. But facts which were part of the inferred closure before must not be removed. In other words: addition of new facts can only monotonically extend the inferred closure.

called *total materialisation*.

Total materialisation is adapted as a reasoning strategy in a number of popular Semantic Web repositories, including some of the standard configurations of Sesame. The TRREE engine - which we elaborated on in section 5.3.2.1 - uses total materialisation.

Bibliography

- [AG] SAP AG. SAP Glossary. http://help.sap.com/saphelp_glossary/en/index.htm.
- [ApMe01] Naresh Apte und Toral Mehta. *Web Services: A Java Developer's Guide Using e-Speak*. Prentice-Hall. November 2001.
- [Bern00] Tim Berners-Lee. Semantic Web Technology Stack. <http://www.w3.org/2000/Talks/1206-xml2k-tbl/slide10-0.html>, 2000.
- [BLHL01] Tim Berners-Lee, James Hendler und Ora Lassila. The Semantic Web. *Scientific American* 284(5), 2001, S. 34–43.
- [BrGu04] D. Brickley und R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. W3c working draft, W3C, February 2004.
- [BrKa04] Jeen Broekstra und Arjohn Kampman. SeRQL: An RDF Query and Transformation Language. Submitted to the International Semantic Web Conference, ISWC, 2004.
- [BrKH02] Jeen Broekstra, Arjohn Kampmann und Frank van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. ISWC 2002, 2002.
- [Broe04] Tom Broens. Context-aware, Ontology based, Semantic Service Discovery. Diplomarbeit, University of Twente, July 2004.
- [BrPSM04] T. Bray, J. Paoli und C.M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. Technischer Bericht, W3C, 2004.
- [Cimi04] Philipp Cimiano. ORAKEL: A Natural Language Interface to an F-Logic Knowledge Base. In *Proceedings of the 9th International Conference on Applications of Natural Language to Information Systems*. Springer, 2004, S. 401–406.
- [Comm] UDDI Spec Technical Committee. UDDI Version 3.0.2. http://www.uddi.org/pubs/uddi_v3.htm.
- [DoMa08] John Domingue und David Martin. Semantic Web Services. Presentation at the ISWC 2008, October 2008.

- [En(O)] Ontology Engineering und Patterns Task Force (OEP). Semantic Web Best Practices and Deployment Working Group. <http://www.w3.org/2001/sw/BestPractices/OEP/>.
- [GATE] GATE - General Architecture for Text Engineering. <http://gate.ac.uk/>.
- [GoGA08] Viji Gobal und N.S. Gowri GAnesh. Ontology Based Search Engine Enhancer. *IAENG International Journal of Computer Science* Band 35:3, 2008.
- [Grad08] Evaluating Semantic Web Service Matchmaking Effectiveness Based on Graded Relevance. Technischer Bericht, Institute of Computer Science, Friedrich-Schiller-University Jena, 2008.
- [HBEV04] Peter Haase, Jeen Broekstra, Andreas Eberhart und Raphael Volz. A Comparison of RDF Query Languages. In *3rd International Semantic Web Conference*, 2004.
- [HKRS⁺04] Matthew Horridge, Holger Knublauch, Alan Rector, Robert Stevens und Chris Wroe. A Practical Guide To Building OWL Ontologies Using The Protégé-OWL Plugin and CO-ODE Tools. <http://www.co-ode.org/resources/tutorials/ProtegeOWLTutorial.pdf>, August 2004.
- [HKRS08] Pascal Hitzler, Markus Krötzsch, Sebastian Rudolph und York Sure. *Semantic Web*. Springer. 2008.
- [HLMS08] Martin Hepp, Pieter De Leenheer, Aldo de Moor und Yor Sure. *Ontology Management - Semantic Web, Semantic Web Services, and Business Applications*. Springer-Verlag GmbH. 2008.
- [Info05] InfoWorld.com. Microsoft, IBM, SAP discontinue UDDI registry effort. http://www.infoworld.com/article/05/12/16/HNuddishut_1.html, December 2005.
- [KüLKR07] Ulrich Küster, Holger Lausen und Brigitta König-Ries. Evaluation of Semantic Service Discovery - A Survey and Directions for Future Research. In *WEWST*, 2007.
- [LeUM06] Yuanguai Lei, Victoria Uren und Enrico Motta. SemSearch: A Search Engine for the Semantic Web. In *3rd International Semantic Web Conference*, 2006.
- [LiHo03] Lei Li und Ian Horrocks. A software framework for matchmaking based on semantic web technology. In *Twelfth International World Wide Web Conference (WWW 2003)*, 2003.
- [LUMP07] Vanessa Lopez, Victoria Uren, Enrico Motta und Michele Pasin. AquaLog: An ontology-driven question answering system for organizational semantic intranets. *Journal of Web Semantics* Band 5, 2007, S. 72–105.

- [Nais88] John Naisbitt. *Megatrends: Ten New Directions Transforming Our Lives*. Grand Central Publishing. 1988.
- [OASI] OASIS. OASIS: Advancing open standards for the global information society. http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=uddi-spec.
- [ODP] Ontology Design Patterns. <http://ontologydesignpatterns.org>.
- [O'EH02] Justin O'Sullivan, David Edmond und Arthur H. M. ter Hofstede. Service Description: A survey of the general nature of services, April 2002.
- [Onto07] Ontotext. OWLIM - Semantic Repository for RDF(S) and OWL. <http://www.ontotext.com/owlim/OWLIMSysDoc.pdf>, September 2007.
- [PKCH05] Jyotishman Pathak, Neeraj Koul, Doina Caragea und Vasant G Honavar. A Framework for Semantic Web Services Discovery. *WIDM*, November 2005.
- [PKPS02] Massimo Paolucci, Takahiro Kawamura, Terry R. Payne und Katia Sycara. Semantic Matching of Web Services Capabilities. In: First International Semantic Web Conference (ISWC2002), 2002.
- [RSNK⁺04] Alan Rector, Guus Schreiber, Natalya F. Noy, Holger Knublauch und Mark A. Musen. Ontology Design Patterns and Problems: Practical Ontology Engineering using Protege-OWL. <http://iswc2004.semanticweb.org/CFParticipation/T2.html>, 2004.
- [RuNo95] S. Russel und P. Norvig. *Artificial Intelligence - A Modern Approach*. Prentice-Hall. 1995.
- [(see)] Michal Zaremba (seekda!). The Fairytale of UDDI Registry and Public Web Services. <http://seekda.com/blog/the-fairytale-of-uddi-registry-and-public-web-services/>.
- [StGA07] Rudi Studer, Stephan Grimm und Andreas Abecker. *Semantic Web Services Concepts, Technologies, and Applications*. Springer-Verlag Gmbh. May 2007.
- [ThSN03] Uwe Thaden, Wolf Siberski und Wolfgang Nejdl. A Semantic Web based Peer-to-Peer Service Registry Network. 2003.
- [Toch] Eran Toch. OPOSSUM - A Web based search engine. <http://dori.technion.ac.il/>.
- [ToGa07] Eran Toch und Avigdor Gal. A Semantic Approach to Approximate Service Retrieval. *ACM Transactions on Internet Technology*(Vol. 8, No.1, Article 2), November 2007.

- [TsAH06] Vassileios Tsetsos, Christos Anagnostopoulos und Stathes Hadjiefthymiades. On the evaluation of semantic web service matchmaking systems. In *4th IEEE European Conference on Web Services (ECOWS2006)*, 2006.
- [VrSu07] Denny Vrandecic und York Sure. How to design better ontology metrics. In *Proceedings of the 4th European Semantic Web Conference (ESWC'07)*. Springer, 2007.
- [W3Ca] W3C. SOAP Version 1.2. <http://www.w3.org/TR/soap/>.
- [W3Cb] W3C. SPARQL Query Language for RDF. <http://www.w3.org/TR/rdf-sparql-query/>.
- [W3C01] W3C. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl#_service, March 2001.
- [W3C03] W3C. Web Services Architecture. <http://www.w3.org/TR/2003/WD-ws-arch-20030808/>, August 2003.
- [W3C04] W3C. OWL Web Ontology Language Reference. <http://www.w3.org/TR/owl-ref>, 2004.
- [Word] WordNet - a lexical database for the English language. <http://wordnet.princeton.edu/>.
- [Zalt08] Philipp Zaltenbach. Automatische Erstellung von semantischen Beschreibungen für SAP Enterprise Services. Diplomarbeit, University of Karlsruhe (TH), August 2008.